Towards Scalable Dataframe Systems

Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo
Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, Aditya Parameswaran
UC Berkeley
{devin.petersohn, smacke, dorx, williamma, dorislee, xmo, jegonzal, hellerstein, adj, adityagp} @berkeley.edu

ABSTRACT

Dataframes are a popular abstraction to represent, prepare, and analyze data. Despite the remarkable success of dataframe libraries in R and Python, dataframes face performance issues even on moderately large datasets. Moreover, there is significant ambiguity regarding dataframe semantics. In this paper we lay out a vision and roadmap for scalable dataframe systems. To demonstrate the potential in this area, we report on our experience building MODIN, a scaled-up implementation of the most widely-used and complex dataframe API today, Python's `pandas`. With pandas as a reference, we propose a simple data model and algebra for dataframes to ground discussion in the field. Given this foundation, we lay out an agenda of open research opportunities where the distinct features of dataframes will require extending the state of the art in many dimensions of data management. We discuss the implications of signature dataframe features including flexible schemas, ordering, row/column equivalence, and data/metadata fluidity, as well as the piecemeal, trial-and-error-based approach to interacting with dataframes.

1. INTRODUCTION

For all of their commercial successes, relational databases have notable limitations when it comes to "quick-and-dirty" exploratory data analysis (EDA) [74]. Data needs to be defined schema-first before it can be examined, data that is not well-structured is difficult to query, and any query beyond `SELECT *` requires an intimate familiarity with the schema, which is particularly problematic for wide tables. For more complex analyses, the declarative nature of SQL makes it awkward to develop and debug queries in a piecewise, modular fashion, conflicting with best practices for software development. In part thanks to these limitations, SQL is often not the tool of choice for data exploration. As an alternative, programming languages such as Python and R support the so-called *dataframe* abstraction. Dataframes provide a functional interface that is more tolerant of unknown data structure and well-suited to developer and data scientist workflows, including REPL-style imperative interfaces and data science notebooks [60].

Dataframes have several characteristics that make them an appealing choice for data exploration:

- an intuitive data model that embraces an implicit ordering on both rows and columns and treats them symmetrically;
- a query language that bridges a variety of data analysis modalities including relational (e.g., filter, join), linear algebra (e.g., transpose), and spreadsheet-like (e.g., pivot) operators;
- an incrementally composable query syntax that encourages easy and rapid validation of simple expressions, and their iterative refinement and composition into complex queries; and
- native embedding in a host language such as Python with familiar imperative semantics.

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 2020, been downloaded over 300 million times, served as a dependency for over 222,000 repositories in GitHub, and starred on GitHub more 25,000 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 from dozens of contributors, each providing operators and their implementations to the DataFrame API to satisfy immediate or ad-hoc needs, spanning capabilities that mimic relational algebra, linear algebra, and spreadsheet computation. To date, the pandas DataFrame API has ballooned to over 200 operators [13]. R, which is both more mature and more carefully curated, has only 70 operators—but this still far more than, say, relational and linear algebra combined [14].

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 a considerable burden on users to select the optimal way of expressing their goal. For example, one blog post cites five different ways to express the same goal, with performance varying from 0.3ms to 600ms (a 1700× increase) [6]; meanwhile, the pandas documentation itself offers multiple recommendations for how to enhance performance [10]. As a result, many users eschew the bulk of the API, relying only on a small subset of operators [12]. The complexity of the API and evaluation semantics also make it difficult to apply traditional query optimization techniques. Indeed, each operator within a pandas "query plan" is executed completely before subsequent operators are executed, with limited optimization, and no reordering of operators or pipelining (unless explicitly done so by the user using `.pipe`). Moreover, the performance of the `pandas.DataFrame` API breaks down when processing even moderate volumes of data that do not fit in memory, as we will see subsequently—this is especially problematic due to pandas' eager evaluation semantics, wherein intermediate data items often surpass main memory limits and must be paged to disk.

To address pandas' scalability challenges, we developed MODIN (`github.com/modin-project/modin`), our first attempt at a scalable dataframe system, which employs parallel query execution to enable unmodified pandas code to run more efficiently on large dataframes. MODIN is used by over 60 downstream projects, and has over 250 forks and 4,800 stars on GitHub in its first 20 months, indicating the impact and need for such systems. MODIN rewrites pandas API calls into a sequence of operators in a new, compact dataframe algebra. MODIN then leverages simple parallelization and a new physical representation to speed up the execution of these operators, by up to **30**× in certain cases, and is able to run to completion on datasets **25**× larger than pandas in others.

Our initial optimizations in MODIN are promising, but only scratch the surface of what's possible. Given that first experience and the popularity of the results, we believe there is room for a ***broad, community research agenda on making dataframe systems scalable and efficient***, with many novel research challenges. Our original intent when developing MODIN was to adapt standard relational database techniques to help make dataframes scalable. However, while the principles (such as parallelism) do apply, their instantiation in the form of specific techniques often differ, thanks to the differences between the data models and algebra of dataframes and relations. Therefore, a more principled foundation for dataframes is needed, comprising a formal data model and an expressive and compact algebra. We describe our first attempt at such a formalization in Section 4. Then, armed with our data model and algebra, we outline a number of research challenges organized around unique dataframe characteristics and the unique ways in which they are processed.

In Section 5, we describe how the dataframe data model and algebra result in new scalability challenges. Unlike relations, dataframes have a flexible schema and are lazily typed, requiring careful maintenance of metadata, and avoidance of the overhead of type inference as far as possible. Dataframes treat rows and columns as equivalent, and metadata (column/row labels) and data as equivalent, requiring flexible ways to keep track of metadata and orientation, placing new metadata awareness requirements on dataframe query planners to avoid physically transposing data where possible. In addition, dataframes are ordered—and dataframe systems often enforce a strict coupling between logical and physical layout; we identify several opportunities to deal with order in a more light-weight, decoupled, and lazy fashion. Finally, the new space of operators—encompassing relational, linear algebra, and spreadsheet operators—introduce new challenges in query processing and optimization.

In Section 6, we describe new challenges and opportunities that emerge from how dataframes are used for data exploration. Unlike SQL which offers an all-or-nothing query modality, dataframe queries are constructed one operator at a time, with ample think-time between query fragments. This makes it more challenging to perform query optimization wherein operators can be reordered for higher overall efficiency. At the same time, the additional thinking time between steps can be exploited to do background processing. Users often inspect intermediate dataframe results of query fragments, usually for debugging, which requires a costly materialization after each step of query processing. However, users are only shown an ordered prefix or suffix of this intermediate dataframe as output, allowing us to prioritize the execution to return this portion quickly and defer the execution of the rest. Finally, users often revisit old processing steps in an ad-hoc process of trial-and-error data exploration. We can consider opportunities to minimize redundant computation for operations completed previously.

Outline and Contributions. In this paper, we begin with an example dataframe workflow capturing typical dataframe capabilities and user behaviors. We then describe our experiences with MODIN (Section 3). We use MODIN to ground our discussion of the research challenges. We ***(i) provide a candidate formalism for dataframes*** and enumerate their capabilities with a new algebra (Section 4). We then outline research challenges and opportunities ***to build on our formalism and make dataframe systems more scalable,*** by optimizing and accounting for ***(ii) the unique characteristics of the new data model and algebra*** (Section 5), as well as ***(iii) the unique ways in which dataframes are used in practice for data exploration*** (Section 6). We draw on tools and techniques from the database research literature throughout and discuss how they might be adapted to meet novel dataframe needs.

In describing the aforementioned challenges, we focus on the pandas dataframe system [13] for concreteness. Pandas is much more popular than other dataframe implementations, and is therefore well worth our effort to study and optimize. We discuss other dataframe implementations and related work in Section 7.

2. DATAFRAME EXAMPLE

In Figure 1, we show the steps taken in a typical workflow of an analyst exploring the relationship between various features of different iPhone models in a Jupyter notebook [60].

Data ingest and cleaning. Initially, the analyst reads in the iPhone comparison chart using `read_html` from an e-commerce webpage, as shown in R1 in Figure 1. The data is verified by printing out the first few lines of the dataframe `products`. (`products.head()` is also often used.) Based on this preview of the dataframe, the analyst identifies a sequence of actions for cleaning their dataset:

- C1 [Ordered point updates]: The analyst fixes the anomalous value of 120MP for Front Camera for the iPhone 11 Pro to 12MP, by performing a point update via `iloc`, and views the result.
- C2 [Matrix-like transpose]: To convert the data to a relational format, rather than one meant for human consumption, the analyst transposes the dataframe (via `T`) so that the rows are now products and columns features, and then inspects the output.
- C3 [Column transformation]: The analyst further modifies the dataframe to better accommodate downstream data processing by changing the column "Wireless Charging" from "Yes/No" to binary. This is done by updating the column using a user-defined `map` function, followed by displaying the output.
- C4 [Read Excel]: The analyst loads price/rating information by reading it from a spreadsheet into `prices` and then examines it.

Analysis. Then, the analyst performs the following operations to analyze the data:

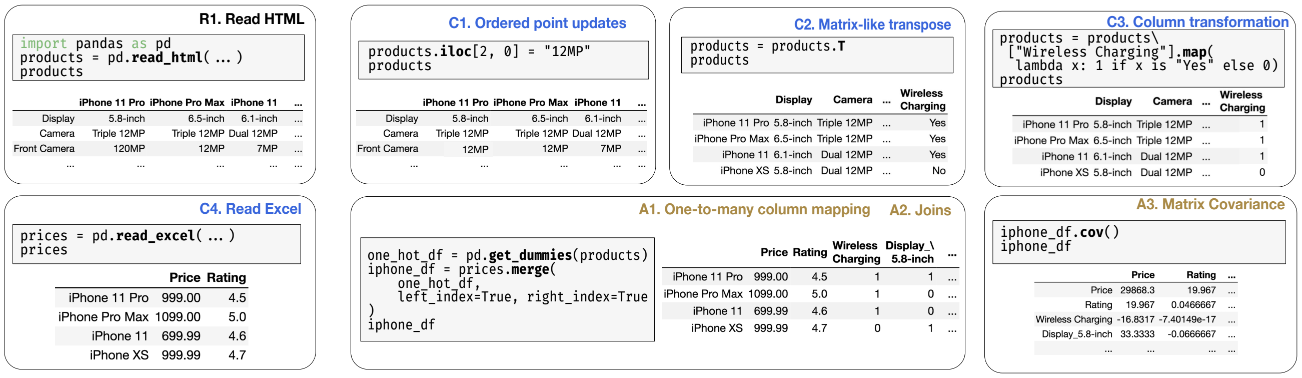
- A1 [One-to-many column mapping]: The analyst encodes non-numeric features in a one-hot encoding scheme via the `get_dummies` function.
- A2 [Joins]: The iPhone features are joined with their corresponding price and rating using the `merge` function. The analyst then verifies the output.
- A3 [Matrix Covariance]: With all the relevant numerical data in the same dataframe, the analyst computes the covariance between the features via the `cov` function, and examines the output.

This example demonstrated only a sample of the capabilities of dataframes. Nevertheless, it serves to illustrate the common use cases for dataframes: immediate visual inspection after most operations, each incrementally building on the results of previous ones, point and batch updates via user-defined functions, and a diverse set of operators for wrangling, preparing, and analyzing data.

3. THE MODIN DATAFRAME SYSTEM

While the pandas API is convenient and powerful, the underlying implementation has many scalability and performance problems. We therefore started an effort to develop a "drop-in" replacement for the pandas API, MODIN[1], to address these issues. In the style of embedded database systems [41, 62], Modin is a library that runs in the same process as the application that imports it. We briefly

[1]MODIN's name is derived from the Korean word for "every", as it targets every dataframe operator.



R1. Read HTML

```
import pandas as pd
products = pd.read_html( ... )
products
```

	iPhone 11 Pro	iPhone Pro Max	iPhone 11	...
Display	5.8-inch	6.5-inch	6.1-inch	...
Camera	Triple 12MP	Triple 12MP	Dual 12MP	...
Front Camera	120MP	12MP	7MP	...
...	...	...	...	...

C1. Ordered point updates

```
products.iloc[2, 0] = "12MP"
products
```

	iPhone 11 Pro	iPhone Pro Max	iPhone 11	...
Display	5.8-inch	6.5-inch	6.1-inch	...
Camera	Triple 12MP	Triple 12MP	Dual 12MP	...
Front Camera	12MP	12MP	7MP	...
...	...	...	...	...

C2. Matrix-like transpose

```
products = products.T
products
```

	Display	Camera	...	Wireless Charging
iPhone 11 Pro	5.8-inch	Triple 12MP	...	Yes
iPhone Pro Max	6.5-inch	Triple 12MP	...	Yes
iPhone 11	6.1-inch	Dual 12MP	...	Yes
iPhone XS	5.8-inch	Dual 12MP	...	No

C3. Column transformation

```
products = products\
 ["Wireless Charging"].map(
  lambda x: 1 if x is "Yes" else 0)
products
```

	Display	Camera	...	Wireless Charging
iPhone 11 Pro	5.8-inch	Triple 12MP	...	1
iPhone Pro Max	6.5-inch	Triple 12MP	...	1
iPhone 11	6.1-inch	Dual 12MP	...	1
iPhone XS	5.8-inch	Dual 12MP	...	0

C4. Read Excel

```
prices = pd.read_excel( ... )
prices
```

	Price	Rating
iPhone 11 Pro	999.00	4.5
iPhone Pro Max	1099.00	5.0
iPhone 11	699.99	4.6
iPhone XS	999.99	4.7

A1. One-to-many column mapping **A2. Joins**

```
one_hot_df = pd.get_dummies(products)
iphone_df = prices.merge(
    one_hot_df,
    left_index=True, right_index=True
)
iphone_df
```

	Price	Rating	Wireless Charging	Display_\ 5.8-inch	...
iPhone 11 Pro	999.00	4.5	1	1	...
iPhone Pro Max	1099.00	5.0	1	0	...
iPhone 11	699.99	4.6	1	0	...
iPhone XS	999.99	4.7	0	1	...

A3. Matrix Covariance

```
iphone_df.cov()
iphone_df
```

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

Figure 1: Example of an end-to-end data science workflow, from data ingestion, preparation, wrangling, to analysis.

describe the challenges we encountered and the lessons we learned during our implementation in Section 3.1, followed by a preliminary case of MODIN's performance in Section 3.2. Finally, we describe MODIN's architecture and implementation.

3.1 Modin Engineering Challenges

When we started our effort to make pandas more scalable, we identified that while many operations in pandas are fast, they are limited by their single-threaded implementation. Therefore, our starting point for MODIN was to add multi-core capabilities and other simple performance improvements to enable pandas users to run their same unmodified workflows both faster and on larger datasets. However, we encountered a number of engineering challenges.

Massive API. The pandas API has over 240 distinct operators, making it challenging to individually optimize each one. After manually trying to parallelize each operator within MODIN, we tried a different approach. We realized that there is a lot of redundancy across these 240 operators. Most of these operators can be rewritten into an expression composed using a much smaller set of operators. We describe our compact set of dataframe operators—our working dataframe algebra—in Section 4.3. Currently, MODIN supports over 85% of the `pandas.DataFrame` API, by rewriting API calls into our working algebra, allowing us to avoid duplicating optimization logic as much as possible. The operators we prioritized were based on an analysis of over 1M Jupyter notebooks discussed in Section4.6. Specifically, we targeted all the functionality in `pandas.DataFrame`, `pandas.Series`, and pandas utilities (e.g., `pd.concat`). To use MODIN instead of pandas, users can simply invoke "`import modin.pandas`", instead of "`import pandas`", and proceed as they would previously. MODIN is implemented in Python using over 30,000 lines of code. MODIN is completely open source and can be found at `https://github.com/modin-project/modin`.

Parallel execution. Since most pandas operators are single-threaded, we looked towards parallelism as a means to speed up execution. Parallelization is commonly used to improve performance in a relational context due to the embarrassingly parallel nature of relational operators. Dataframes have a different set of operators than relational tables, supporting relational algebra, linear algebra, and spreadsheet operators, as we saw in Section 2, and we will discuss in Section 4. We implemented different internal mechanisms for exploiting parallelism depending on the data dimensions and operations being performed. Some operations are embarrassingly parallel and can be performed on each row independently (e.g., C3 in Figure 1), while others (e.g., C2, A1, A3) cannot. To address the challenge of differing levels of parallelism across operations, we designed MODIN to be able to flexibly move between common partitioning schemes: row-based (i.e., each partition has a collection of rows), column-based (i.e., each partition has a collection of columns), or block-based partitioning (i.e., each partition has a subset of rows and columns), depending on the operation. Each partition is then processed independently by the execution engine, with the results communicated across partitions as needed.

Supporting billions of columns. While parallelism does address some of the scalability challenges, it fails to address a major one: the ability to support tables with billions of columns—something even traditional database systems do not support. Using the pandas API, however, it is possible to transpose a dataframe (as in Step C2) with billions of rows into one with billions of columns. In many settings, e.g., when dealing with graph adjacency matrices in neuroscience or genomics, the number of rows and number of columns can both be very large. For these reasons, MODIN treats rows and columns essentially equivalently, a property of dataframes will discuss in detail in Section 4. In particular, to transpose a large dataframe, MODIN employs block-based partitioning, where each block consists of a subset of rows and columns. Each of the blocks are individually transposed, followed by a simple change of the overall metadata tracking the new locations of each of the blocks. The result is a transposed dataframe that does not require any communication.

3.2 Preliminary Case Study

To understand how the simple optimizations discussed above impact the scalability of dataframe operators, we perform a small case study evaluating MODIN's performance against that of pandas using microbenchmarks on an EC2 x1.32xlarge (128 cores and 1,952 GB RAM) node using a New York City taxicab dataset [56] that was replicated 1 to 11 times to yield a dataset size between 20 to 250 GB, with up to 1.6 billion rows. We consider four queries:

- map: check if each value in the dataframe is null, and replace it with a TRUE if so, and FALSE if not.
- groupby (n): group by the non-null "passenger_count" column and count the number of rows in each group.
- groupby (1): count the number of non-null rows in the dataframe.
- transpose: swap the columns and rows of the dataframe and apply a simple (map) function across the new rows.

We highlight the difference between group by with one group and n groups, because with n groups data shuffling and communication are a factor in performance. With groupby(1), the communication overheads across groups are non-existent. We include transpose to demonstrate that MODIN can handle data with billions of columns. This query also shows where pandas crashed or did not complete in more than 2 hours.

Figure 2 shows that for the group by (n) and group by (1) operations, MODIN yields a speedup of up to 19$\times$ and 30$\times$ relative to pandas, respectively. For example, a group by (n) on a 250GB dataframe, pandas takes about 359 seconds and MODIN takes 18.5 seconds, a speedup of more than 19$\times$. For map operations, MODIN

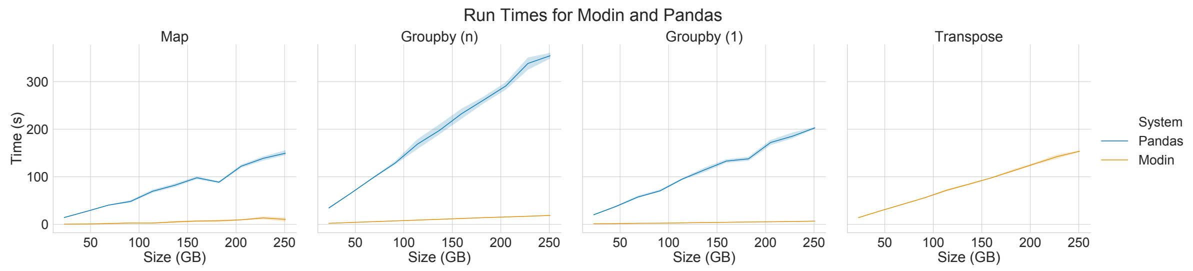


Figure 2: For each function, we show the runtime for both MODIN and pandas and the 95% confidence interval. There are no times for transpose with pandas as pandas is unable to run transpose beyond 6 GB.

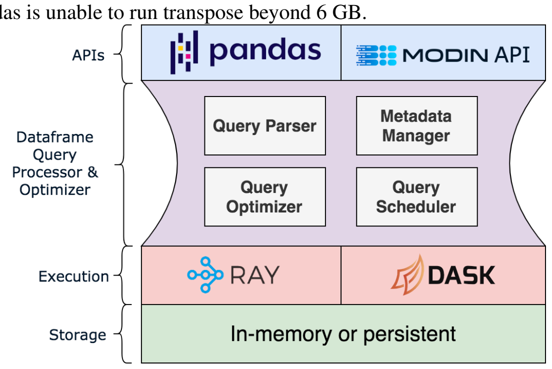


Figure 3: MODIN architecture.

is about 12× faster than pandas. These performance gains come from simple parallelization of operations within MODIN, while pandas only uses a single core. During the evaluation of transpose, pandas was unable to transpose even the smallest dataframe of 20 GB (∼150 million rows) after 2 hours. Through separate testing, we observed that pandas can only transpose dataframes of up to 6 GB (∼6 million rows) on the hardware we used for testing.

Takeaways. Our preliminary case study and our experience with MODIN demonstrates the promise of integrating simple optimizations to make dataframe systems scalable. Next, we define a dataframe data model and algebra to allow us to ground our subsequent discussion of our research agenda, targeting the unique characteristics of dataframes and the unique ways in which they are used. We defer further performance analyses of MODIN to future work.

3.3 The MODIN Architecture

MODIN's architecture is *modular* for easy integration of new storage and execution engines, APIs, and optimizations. It consists of four layers: the API layer, the query processing and optimization layer, the execution layer, and the storage layer, shown in Figure 3.

API layer. Users can leverage MODIN via a pandas-based API, or directly via a leaner and simpler MODIN API based on the algebra in Section 4.3. In either case, the API layer translates each call into a dataframe algebraic expression, and passes that to the next layer for execution. The layer isolates users from changes to the layers below, while allowing users to leverage the API modality they are most comfortable with. Future implementations may support other user APIs for working with dataframes, such as SQL or relational algebra. Our pandas-based API currently supports about 150 of over 200 pandas dataframe APIs, and rewrites each of them into dataframe algebraic expressions.

Query processing and optimization layer. As shown in Figure 3, the query processing layer follows a "narrow waist" design, exposing a small API based on the dataframe algebra, and implements the data model from Section 4.2. This layer parses, optimizes, and executes dataframe queries with the help of layers below. As we will describe in Section 3.1, MODIN leverages parallel execution of dataframe queries on multiple dataframe partitions, scheduled on execution engines in the next layer. This layer also keeps track of dataframe metadata including row labels, column labels, and column data types. Recall that data types may not be specified on dataframe creation, so MODIN induces types on-the-fly (using the S function) when needed for a specific operation.

Execution layer. MODIN supports distributed processing of dataframe partitions using two execution frameworks: Ray [53] and Dask [31]. Both Ray and Dask are task-parallel asynchronous execution engines exposing an API that requires defining a task or function and providing data for the task to run on. Integration of a new execution framework is simple, often requiring fewer than 400 lines of code.

Storage layer. MODIN's modular storage layer supports both main memory and persistent storage out-of-core (also called memory spillover), allowing intermediate dataframes to exceed main-memory limitations while not throwing memory errors, unlike pandas. To maintain pandas semantics, the dataframe partitions are freed from persistent storage once a session ends.

4. DATAFRAME FUNDAMENTALS

There are many competing open-source and commercial implementations of dataframes, but there is no formal definition or enumeration of dataframe properties in the literature to date. We therefore propose a formal definition of dataframes to allow us to describe our subsequent research challenges on a firm footing, and also to provide background to readers who are unfamiliar with dataframes. In this section, we start with a brief history (Section 4.1), and provide a reference data model (Section 4.2) and algebra (Section 4.3) to ground discussion. We then demonstrate the expressiveness of the algebra via a case study (Section 4.4) and discuss extensions (Section 4.5). We finally provide some quantitative statistics into dataframe usage in Section 4.6.

4.1 A Brief History of Dataframes

The S programming language was developed at Bell Laboratories in 1976 to support statistical computation. Dataframes were first introduced to S in 1990, and presented by Chambers, Hastie, and Pregibon at the Computational Statistics conference [27]. The authors state: "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 ..." Chambers and Hastie then extended this paper into a 1992 book [28], which states "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." and "... 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."

The R programming language, an open-source implementation of S with some additional innovations, was first released in 1995, with a stable version released in 2000, and gained instant adoption among the statistics community. Finally, in 2008, Wes McKinney developed pandas in an effort to bring dataframe capabilities with R-like semantics to Python, which as we described in the introduction, is now incredibly popular. In fact, pandas is often cited as the reason for Python's popularity [7,9], now surpassing Java and C++ [8]. We discuss other dataframe implementations in Section 7.

4.2 Dataframe Data Model

As Chambers and Hastie themselves state, dataframes are not familiar mathematical objects. Dataframes are not quite relations, nor are they matrices or tensors. In our definitions we borrow textbook relational terminology from Abiteboul, et al. [17, Chapter 3] and adapt it to our use.

The elements in the dataframe come from a known set of domains $Dom = \{\mathbf{dom}_1, \mathbf{dom}_2, ...\}$. For simplicity, we assume in our discussion that domains are taken from the set $Dom = \{\Sigma^*$, **int**, **float**, **bool**, **category**$\}$, though a few other useful domains like datetimes are common in practice. The domain Σ^* is the set of finite strings over an alphabet Σ, and serves as a default, uninterpreted domain; in some dataframe libraries it is called **Object**. Each domain contains a distinguished *null* value, sometimes written as NA. Each domain $\mathbf{dom}_i$ also includes a *parsing function* $p_i : \Sigma^* \rightarrow \mathbf{dom}_i$, allowing us to interpret the values in dataframe cells as domain values (including possibly *null*).

A key aspect of a dataframe is that the domains of its columns may be induced from data *post hoc*, rather than being declared *a priori* as in the relational model. We define a **schema induction function** $S : \Sigma^* \rightarrow Dom$ that assigns an array of m strings to a domain in Dom. This schema induction function is applied to a given column and returns a domain that describes this array of strings; we will return to this function later.

Armed with these definitions, we can now define a dataframe:

Definition 4.1. A **dataframe** is a tuple (A_{mn}, R_m, C_n, D_n), where A_{mn} is an array of entries from the domain Σ^*, R_m is a vector of row labels from Σ^*, C_n is a vector of column labels from Σ^*, and D_n is a vector of n domains from 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 $S(\cdot)$ to the corresponding column of A_{mn} to get its domain i and then $p(\cdot)$ to get its values.

We depict our conceptualization of dataframes in Figure 4. In our example of Figure 1, dataframe products after step R1 has R_m corresponding to an array of labels [Display, Camera, ...]; C_n corresponding to an array of labels [iPhone 11 Pro, iPhone Pro Max, ...]; A_{mn} corresponding to the matrix of values beginning with 5.8-inch, with $m = 6, n = 4$. Here, D_n is left unspecified, and may be inferred using $S(\cdot)$ per column to possibly correspond to $[\Sigma^*, \Sigma^*, \Sigma^*, \Sigma^*]$, since each of the columns contains strings.

Rows and columns are symmetric in many ways in dataframes. Both can be referenced explicitly, using either numeric indexing (positional notation) or label-based indexing (named notation). In our example in Figure 1, the products dataframe is referenced using positional notation in step C1 with products.iloc[2, 0] to modify the value in the third row and first column, and by named notation in step C3 using products ["Wireless Charging"] to modify the column corresponding to "Wireless Charging". The relational model traditionally provides this kind of referencing only for columns. Note that row position is exogenous to the data—it *need not* be correlated in any way to the data values, unlike sort orderings found in relational extensions like SQL's ORDER BY clause. The positional notation allows for (row, col) references to index individual values, as is familiar from matrices.

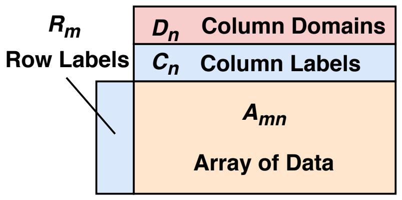


Figure 4: The Dataframe Data Model

A subtler distinction is that row and column labels are from the same set of domains as the underlying data (Dom), whereas in the traditional relational model, column names are from a separate domain (called **att** [17]). This is important to point out because there are dataframe operators that copy data values into labels, or copy labels into data values, discussed further in Section 4.3.

One distinction between rows and columns in our model is that columns have a schema, but rows do not. Said differently, we parse the value of any cell based on the domain of its column. We can also imagine an orthogonal view, in which we define explicit schemas (or use a schema induction function) on rows, and a corresponding row-wise parsing function for the cells. In our formalism, this is achieved by an algebraic operator to transpose the table and treat the result column-wise (Section 4.3). By restricting the data model to a single axis of schematization, we provide a simple unique interpretation of each cell, yet preserve a flexibility of interpretation in the algebra. In Sections 5.1.2 and 5.2.2 we return to the performance and programming implications of programs that make use of schemas on a dataframe and its transpose (i.e. "both axes").

When the schema D_n has the same domain **dom** for all n columns, we call this a *homogeneous* dataframe, and its rows and columns can be considered symmetrically to have the domain **dom** differing only in dimension. As a special case, consider a homogeneous dataframe with a domain like **float** or **int** and operators $+, \times$ that satisfy the algebraic definition of a field. We call this a *matrix* dataframe, since it has the algebraic properties required of a matrix, and can participate in linear algebra operations simply by parsing its values and ignoring its labels. The dataframe iphone_df after step A2 in Figure 1 is one such example; thus it was possible to perform the covariance operation in step C3. Matrix dataframes are commonly used in machine learning pipelines.

Overall, while dataframes have roots in both relational and linear algebra, they are neither tables nor matrices. Specifically, when viewed from a relational viewpoint, the dataframe data model differs in the following ways:

Dataframe Characteristic	Relational Characteristic
Ordered table	Unordered table
Named rows labels	No naming of rows
A lazily-induced schema	Rigid schema
Column names from $d \in Dom$	Column names from att [17]
Column/row symmetry	Columns and rows are distinct
Support for linear alg. operators	No native support

And when viewed from a matrix viewpoint, the dataframe data model differs in the following ways:

Dataframe Characteristic	Matrix Characteristic
Heterogeneously typed	Homogeneously typed
Both numeric and non-numeric types	Only numeric types
Explicit row and column labels	No row or column labels
Support for rel. algebra operators	No native support

We will exploit these two viewpoints in our dataframe algebra to allow us to define both relational and linear algebra operations. Due to these differences, a new body of work will be needed to support the scale required for modern data science workflows.

Data model Comparisons

Before we go on, we address some key distinctions between dataframes and other familiar data models.

Comparison with matrices. All matrices can be represented as dataframes (with null labels). Not all dataframes can be matrices, however, even if we strip off their labels! Matrices are homogeneous in schema, but dataframes allow for schemas with multiple domains. Even if we ignore the schema, a dataframe is still not a matrix—opaque strings from Σ^* do not satisfy the properties of a field as required by a matrix.

Comparison with relational tables. A relation is defined by a declared schema, and there are many possible *instances* of a relation—sets of tuples that satisfy the schema. An instance can be thought of as a fixed relational table. Dataframes are something like relation instances: they represent a fixed set of data. However their schema can be unspecified and hence *induced* based on their content by a schema induction function S. This flexibility is critical to dataframes.

Moreover, dataframes impose an ordering and naming on their rows. Object-oriented relational extensions such as the Postgres data model also introduced implicit row identifiers [66], but typically relational models do not impose a row ordering. Of course, we can capture this semantics in a relational model via design discipline: we can ensure that all our relations have a unique key (for naming), and an ordering key (for ordering), which are exogenous to the actual data columns in the table. In this sense, all dataframes can be represented as relation instances conforming to some (potentially induced) schema with appropriate keys.

Even so, a key difference between dataframes and relations is the symmetry between rows and columns. This aspect, along with the freedom to induce a schema on a per-instance basis, make it possible to define a transpose operator on dataframes. Since the underlying representation is uninterpreted, we are free to induce a different relational schema after transposition of a dataframe.

Comparison with spreadsheets. At a high level, a spreadsheet is an array of heterogeneously-typed cells that may contain strings or formulae. A spreadsheet thus stores code as well as data. The strings are dynamically interpreted into a variety of domains. It is tempting to think of formula-free spreadsheets as being similar to dataframes, given the common row/column indexing scheme and the dynamic typing. But in general they are quite different. The data representations possible in spreadsheets are quite free, and in practice often quite irregular. Dense regions of data are often interspersed with empty regions, or with cells containing comments and other forms of human-centric annotation or metadata. Spreadsheets have a notion of a "range"—a subarray or even a set of subarrays—which may be sparsely located in the data grid and represent diverse or unrelated data sets stored in the same spreadsheet. As a result of this freedom of structure, bulk algebraic operations are difficult to define generally within spreadsheets. Some modern spreadsheets allow a range to be labeled as a "table" that is interpreted with a schema and maintains an ordering; further extensions to "pivot tables" allow for row and column labels. Using these constructs it is possible for a spreadsheet to represent one or more dataframes. But the relative simplicity of dataframes enables a much simpler algebra and easier implementation and optimization of the algebra's operators.

4.3 Dataframe Algebra

While developing MODIN, we discovered that there exists a "kernel" of operators that encompasses the massive APIs of pandas and R. We developed this "kernel" into a new dataframe algebra, which we describe here, while explicitly contrasting it with relational algebra. We do not argue that this set of operators is minimal, but we do feel it is both expressive and elegant; we demonstrate via a case study in Section 4.4 can be used to express `pivot`; other examples of rewriting for operators within pandas can be found in our technical report [61]. Based on the contrast with relational algebra, we are in a position to articulate research challenges in optimizing dataframe algebra expressions in subsequent sections.

To the best of our knowledge, an algebra for dataframes has never been defined previously. Recent work by Hutchinson et al. [42, 43] proposes an algebra called Lara that combines linear and relational algebra, exposing only three operators: JOIN, UNION, and `Ext` (also known as "flatmap"); however, the operators below that manipulate metadata would not be possible in Lara without placing the metadata as part of the data. Other differences stem from the flexible data model and lazily induced schema. That said, as we continue to refine our algebra, we will draw on Lara as a reference.

We list the algebra operators we have defined in Table 1: the rows correspond to the operators, and the columns correspond to their properties. The operators encompass ordered analogs of extended relational algebra operators (from SELECTION to RENAME), one operator that is not part of extended relational algebra but is found in many database systems (WINDOW), one operator with that admits independent use unlike in database systems (GROUPBY), as well as four new operators (TRANSPOSE, MAP, TOLABELS, and FROMLABELS). The ordered analogs of relational algebra operators preserve the ordering of the input dataframe(s). If there are multiple arguments, the result is ordered by the first argument first, followed by the second. For example, UNION simply concatenates the two input dataframes in order, while CROSS-PRODUCT preserves a nested order, where each tuple on the left is associated, in order, with each tuple on the right, with the order preserved.

Note that languages choose different approaches to inferring the schema after a TRANSPOSE with important implications for usability. For example, in R, a TRANSPOSE with heterogeneous D_n ends up coercing everything to **string**, which may make it impossible to apply another TRANSPOSE and yield a dataframe equivalent to the original D_n. In Python, everything is coerced to `Object`, which has typing information embedded at runtime, so the schema induction function can always recover the original D_n after two transposes.

Transpose. TRANSPOSE interchanges rows and columns, so that the columns of the dataframe become the rows, and vice-versa. Formally, given a dataframe $DF = (A_{mn}, R_m, C_n, D_n)$, we define TRANSPOSE($DF$) to be a dataframe $(A^T_{nm}, C_n, R_m, null)$, where A^T_{nm} is the array transpose of A_{mn}. Note that the schema of the result may be induced by S, and may not be similar to the schema of the input. TRANSPOSE is useful both for matrix operations on homogenous dataframes, and for data cleaning or for presentation of "crosstabs" data. In step C2 in our example in Figure 1, the table was not oriented properly from ingest, and a transpose was required to give us the desired table orientation.

In pandas and other dataframe implementations, it is possible to perform many operations along either the rows or columns via the `axis` argument. Instead, to minimize redundancy, we define operators on collections of rows, as in relational algebra, and enable operations across columns by first performing a TRANSPOSE, applying the operation, and then a TRANSPOSE again to return to the original orientation. That said, performing TRANSPOSE can be expensive (as we will see in Section 3), so one of our goals will be to postpone performing it or avoid it entirely. Moreover, given the presence of TRANSPOSE in the algebra, we need to be prepared to handle dataframes that are not only extremely high in cardinality ("tall") but also extremely high in arity ("wide").

Operator	(Meta)data		Schema	Origin	Order	Description
SELECTION		$\times$	static	REL	Parent	Eliminate rows
PROJECTION		$\times$	static	REL	Parent	Eliminate columns
UNION		$\times$	static	REL	Parent†	Set union of two dataframes
DIFFERENCE		$\times$	static	REL	Parent†	Set difference of two dataframes
CROSS PRODUCT / JOIN		$\times$	static	REL	Parent†	Combine two dataframes by element
DROP DUPLICATES		$\times$	static	REL	Parent	Remove duplicate rows
GROUPBY		$\times$	static	REL	New	Group identical attribute values for a given (set of) attribute(s)
SORT		$\times$	static	REL	New	Lexicographically order rows
RENAME	$(\times)$		static	REL	Parent	Change the name of a column
WINDOW		$\times$	static	SQL	Parent	Apply a function via a sliding-window (either direction)
TRANSPOSE	$(\times)$	$\times$	dynamic	DF	Parent$^\diamond$	Swap data and metadata between rows and columns
MAP	$(\times)$	$\times$	dynamic	DF	Parent	Apply a function uniformly to every row
TOLABELS	$(\times)$	$\times$	dynamic	DF	Parent	Set a data column as the row labels column
FROMLABELS	$(\times)$	$\times$	dynamic	DF	Parent	Convert the row labels column into a data column

Table 1: Dataframe Algebra. $\dagger$: Ordered by left argument first, then right to break ties. $\diamond$: Order of columns is inherited from order of rows and vice-versa.

In the algebra defined above, we define operators only on collections of rows, as in relational algebra, allowing TRANSPOSE to toggle the axis of application of the operators. Operations along the columns require a TRANSPOSE, application of the desired operator, and a TRANSPOSE again to return to the original orientation. With this flexibility, operators on the dataframe can be performed along either the columns or the rows.

The asymmetry of row and column types in the relational model makes TRANSPOSE impossible to define for relations with non-homogeneous column domains (for which the sets in D_n differ): there is no data-independent way to derive a relational output schema for TRANSPOSE from the input schema. In the dataframe data model, the data-dependent schema induction function provides an output schema.

TRANSPOSE can also be extremely computationally expensive depending on the system architecture and partitioning. In its implementation, it will often be important to postpone the calculation of TRANSPOSE until the last possible moment because of the associated computation costs. Moreover, given the presence of TRANSPOSE in the algebra, we need to be prepared to handle dataframes that are not only extremely high in cardinality ("tall") but also extremely high in arity ("wide").

Map. The map operator takes some function f and applies it to each row individually, returning a single output row of fixed arity. The purpose of the map operator is to alter each dataframe row uniformly. MAP is useful for data cleaning and feature engineering (e.g., step C3 in Figure 1). Given a dataframe $DF = (A_{mn}, R_m, C_n, D_n)$, the result of MAP(DF, f) is a dataframe $(A'_{mn'}, R_m, C'_{n'}, D'_{n'})$ with $f : D_n \to D'_{n'}$, where $A'_{mn'}$ is the result of the function f as applied to each row, $C'_{n'}$ is the resulting column labels, and $D'_{n'}$ is the resulting vector of domains. Notice that in this definition, the number of columns (n') *and* the column labels ($C'_{n'}$) can change based on this definition, but they must be changed uniformly for every row. The vector of domains $D'_{n'}$ may, in many cases, be inferred from the type of the function f.

Extended relational algebra supports map via the use of functions in the subscript of projection operators (i.e., in the SELECT clause of SQL). However, this projection syntax is linear in the arity of the relation, which is cumbersome for very wide schemas (e.g., after a TRANSPOSE). In this definition, MAP is passed an entire row as an argument so it can reason across columns in a generic fashion without enumerating them, whereas SQL expressions (including UDFs) typically require specific fields from the row as scalar arguments. For example, consider a transformation that needs to ensure the values in all **float**-domain columns in a given row sum to 1.0; a generic, reusable MAP function can normalize the value in each **float** field by the sum of the **float** fields in that row; instead, a SQL expression would have to be crafted specially for each schema.

ToLabels. The TOLABELS operator projects one column out of the matrix of data, A_{mn}, to be set as new row labels for the resulting dataframe, replacing the old labels. Given $DF = (A_{mn}, R_m, C_n, D_n)$ and some column label L, TOLABELS(DF, L) returns a dataframe $(A'_{m(n-1)}, L, C'_n, D'_n)$, where C'_n (respectively D'_n) is the result of removing the label L from C_n (respectively D_n). With this capability, data from A_{mn} can be promoted into the metadata of the dataframe and referenced by name during future interactions.

From a relational perspective, this operator is rather unusual in that it converts data into metadata. Dataframe users are interested in wrangling and cleaning data, so operations that let them move entries between metadata and data are popular and convenient to use. In fact, TOLABELS followed by TRANSPOSE is, in effect, promoting data values into column labels, which is impossible using relational operators.

FromLabels. FROMLABELS creates a new dataframe with the row labels inserted into the array A_{mn} as a new column of data at position 0 with a provided column label. The data type of the new column starts as *null* until it can be induced by the schema induction function S. The row labels of the resulting dataframe are set to the default label: the order rank of each row (positional notation). Formally, given a dataframe $DF = (A_{mn}, R_m, C_n, D_n)$ and a new column label L we define FROMLABELS(DF, L) to be a dataframe $(R_m + A_{mn}, P_m, [L] + C_n, [null] + D_n)$, where $R_m + A_{mn}$ is the concatenation of the row labels R_m with the array of data A_{mn}, P_m is the positional notation values for all of the rows: $P_m = (0, ..., m-1)$, and $[L] + C_n$ is the result of prepending the new column label L to the column labels C_n.

GroupBy. As in relational algebra, our GROUPBY operator groups by one or more columns, and aggregates one or more columns together or separately. Unlike relational algebra, where aggregation must result in atomic values, dataframes can support composite values within a cell, allowing a broader class of aggregation functions to be applied. One special function, collect, groups rows with the same grouping attribute values into separate dataframes and returns these as the (composite) aggregate values. Pandas's groupby function has similar behavior and applies collect to the non-grouped attributes, coupled with an implicit TOLABELS call that elevates the grouping attribute values to the row labels. We will use collect in our examples subsequently.

Window. WINDOW-type operations are largely analogous to those used in recent SQL extensions to RDBMSs like PostgreSQL and SQL Server. The key difference is that, in SQL, many windowing functions such as LAG and LEAD require an additional ORDER BY to be well-defined; in dataframe algebra, the inherent ordering already present in dataframes makes such a clause purely optional.

FROMLABELS is the opposite of the TOLABELS operator, and the two of these give the user complete control over moving data to and from the dataframe's labels. This allows users to apply operators on the dataframe's metadata (specifically the row labels), which is particularly useful for operators like JOIN and GROUPBY. Conceptually, this operator also allows the positional notation of the dataframe to be treated as data if multiple FROMLABELS are chained together. However, because the order is immutable, it is impossible to update the order of the dataframe directly in this way. Despite providing the ability to promote data to row labels (named notation), it is impossible in this algebra to promote data to positional notation. If the users wished to reorder the data, they may JOIN with another dataset with a specific order or SORT based on some column(s).

From a relational point of view, FROMLABELS enables the capability to push metadata into the data to be queried and operated on. Thanks to this operator and TOLABELS specifically, column and row labels must be of type Σ^* so that these operators make sense. FROMLABELS also has some interesting interaction with the schema induction function S, where labels can be interpreted as any type in Dom when they are added to the data via FROMLABELS and then operated on. It is important to point that out here in the definition, but we leave the enumeration of the nuances of this interaction to future work.

4.4 Algebra Examples

To demonstrate the expressiveness of the algebra above, we show how it can be used to elegantly and succinctly express pivot, which is particularly challenging in relational databases due to the need for relations to be declared schema-first [30,79]. The flexible schemata inherent in the dataframe data model enables a succinct description of pivot.

To start off, many pandas functions provide essentially identical functionality to dataframe operators, e.g., sort_values for SORT, merge for JOIN, groupby for GROUPBY, append for UNION, reset_index for FROMLABELS, and set_index for TOLABELS. The function transform is a special case of MAP that applies a fixed function to each value within a row, thereby preserving the input arity, while apply is another special case where a fixed function is applied on a per-row-basis to combine values across multiple columns to generate a new column.

A number of pandas functions correspond to dataframe operators, with specific UDFs. As examples for WINDOW, cummax computes the cumulative max of values for one or more columns, diff takes the difference between elements in a column and preceding values, and shift shifts rows down to align with a new row label, maintaining the order of the data. Likewise, for MAP, fill_na converts all null values to another value, isna replaces each value with a boolean based on whether or not they are null, and str.upper converts all the string values to upper case. In fact, pandas has many functions that implement string and date-time transformations.

Finally, there are several pandas functions that are compositions of dataframe operators. We list a few examples below, with informal descriptions on how they may be rewritten using the algebra.

The agg['f1','f2', ...] function in pandas computes aggregate functions f1, f2, ..., for each of the columns individually, with the resulting dataframe containing one row per aggregate, i.e., the first row corresponds to the f1 aggregates, the second to the f2 aggregates, and so on. This function can be rewritten using one GROUPBY operator per aggregate function to produce a single row corresponding to the aggregates, followed by a UNION to append these rows to each other in the order the aggregates are listed. Another approach is to perform a TRANSPOSE, then a MAP to compute all the necessary aggregates, one per column, followed by another TRANSPOSE to bring the result to the right orientation.

The pandas function target.reindex_like(reference) supports changing a given dataframe (the target) by reordering its rows and columns to match those of another dataframe (the reference). This operator is useful for aligning two dataframes for comparison purposes. One way to express this function using dataframe operators would be to first FROMLABELS on both dataframes to allow the row labels to become part of the data, followed by a INNER JOIN between the two dataframes on the row labels, with the reference as the left operand; followed by a MAP to project out the reference dataframe attributes (leaving behind reference's ordering). Finally, TOLABELS can be used to move the row labels back from the data.

Wide Table of MONTHs

Month	2001	2002	2003
Jan	100	150	300
Feb	110	200	310
Mar	120	250	NULL

Narrow Table (SALES)

Year	Month	Sales
2001	Jan	100
2001	Feb	110
2001	Mar	120
2002	Jan	150
2002	Feb	200
2002	Mar	250
2003	Jan	300
2003	Feb	310

Pivot ⟶

⟵ Unpivot

Year	Jan	Feb	Mar
2001	100	110	120
2002	150	200	250
2003	300	310	NULL

Wide Table of YEARs

Figure 5: Pivot table example, reproduced from [30], demonstrating pivoting over two separate columns, "Month" and "Year".

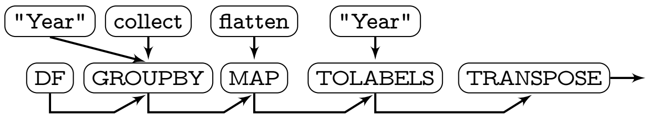


Figure 6: Logical plan for pivoting a dataframe around the "Year" column using the dataframe algebra from this section.

The pivot operator elevates a column of data into the column labels and creates a new dataframe reshaped around these new labels (see Figure 5). The pivot operator has been described and implemented in relational systems [30,79] but it is simpler to express in the algebra from Section 4.3.

Since there is no need to know the names of the new columns or the resulting schema *a priori*, a pivot can be expressed concisely in dataframe algebra as a combination of four operators in the plan shown in Figure 6. Recall that it is possible to elevate data to the column labels by using TOLABELS followed by TRANSPOSE. In this case, the TOLABELS operator would be applied on the label of the column being pivoted over, "Year" in this example. After this step, we perform a GROUPBY on the pivoted attribute, "Year" with a collect aggregation applied to the remaining attributes to produce a per-Year dataframe as a composite aggregated value. This aggregated value is manipulated by a MAP operator with a function that flattens the grouped data into the correct orientation. This results in a table pivoted around the attribute selected for the TOLABELS operator. Notice in Figure 5 that transposing the dataframe labeled "Wide Table in Months" results in the correct data layout for the "Wide Table in Years". This is one example of how TRANSPOSE can be exploited: cost models in dataframe

Algebra Op	Pandas Op	Pandas Op Description
MAP	fillna	Convert null values to another value
	isnull	Determine if elements are null
TRANSPOSE	transpose	Exchange the columns and row
AS_LABELS	set_index	Set the dataframe row labels using existing column(s)
RESET_LABELS	reset_index	Insert the row labels into the dataframe and set row labels to the default

Table 2: pandas operators that directly map to algebra operators.

query optimizers can choose the more efficient pivot column and TRANSPOSE at the end.

To demonstrate the expressivity and power of of this algebra, we demonstrate the real-world application of applying it to pandas. In this section, we demonstrate how a few of the more exotic pandas operators can be written with the algebra presented in Section 4.3. We will illustrate that there are operators which have a one-to-one mapping with the algebra we described in the algebra, and that there are compositions of the algebra operators.

One-to-one mappings are shown in Table 2. pandas has an operator for each of the

Not all of the more than 200 `pandas.DataFrame` methods map one-to-one to the algebra operators we have described. We now describe some of the more interesting and complicated pandas operations and how they can be written in the algebra we have defined.

4.5 Extensions to the Formalism

Our data model so far is quite simple. We now describe a few additional extensions for our data model that do not provide any additional expressive power, but make certain operations more convenient.

Multiple label columns. The data model can, optionally, have multiple row label columns or multiple column label rows. Often, these are presented in a hierarchical or nested manner in pandas. As an example, in a dataframe tabulating sales, we could have two row label rows that are nested, with the first (external) row label row corresponding to the years, and the second (internal) row label row corresponding to the quarters within each year. In our representation, we can simply capture this by repeating the external row label values, and combining the row label columns to give a single composite value, as shown below:

$$\left.\begin{array}{rr} 2017 & Q1 \\ & Q2 \\ & \ldots \\ 2018 & Q1 \\ & \ldots \end{array}\right\| \; A_{mn} \quad \Longrightarrow \quad \left.\begin{array}{l} (2017, Q1) \\ (2017, Q2) \\ \ldots \\ (2018, Q1) \\ \ldots \end{array}\right\| \; A_{mn}$$

Label flexibility and types. Row labels can have a predefined type or domain from *Dom*—this type can be recorded separately and used to augment the schema D_n when performing an AS LABELS operation, thereby avoiding having to induce it using S. Due to the symmetry between columns and rows, column labels also have this constraint. Additionally, labels can have duplicate values or be null; so labels are not like primary keys.

We finally define another notion that will come in handy in future sections: a *dataframe-like* system is one that supports some, but not all dataframe properties as defined in the data model and algebra above. For example, a dataframe-like system might support unordered weakly-typed relations, with queries being composed incrementally over the course of many statements. We return to this notion and provide some example systems in Section 7.

Workflow Definitions.

We now briefly introduce some terms that will allow us to describe how dataframes are manipulated during a data analysis workflow.

Operator. A *dataframe operator*, or simply an *operator*, is an atomic dataframe processing step that takes multiple dataframe arguments and returns a dataframe as a result. We will describe the operators in the context of the dataframe algebra in Section 4.3.

Statement. A *dataframe statement* is an expression composed entirely of dataframe operators and is the unit of interaction between the user and the system. In a notebook environment, a statement corresponds to a single cell; each of which is executed one at a time, as we saw in Figure 1. In an interpreted environment (e.g., iPython), a statement is a single block of code.

Query. A sequence of statements chained together form a *dataframe query*. Following variable references, a query can be represented as a DAG of operators and dataframes, with the input dataframes at the leaves, and the queries as the root(s). A dataframe query is analogous to a SQL query, but it is composed incrementally across many statements.

Session. A *session* is a complete, end-to-end analysis workflow, comprising one or more queries issued across many statements. A session begins when the user starts a notebook or interpreter environment and ends when the user shuts down that environment.

4.6 Dataframe Usage Statistics

To study how dataframe users interact with the pandas API, we analyzed a comprehensive dataset of 1 million Jupyter notebooks hosted on github.com from Rule et al. [68]. Out of the 1 million Jupyter notebooks, about 40% used pandas. We used the `jupyter nbconvert` module to convert each notebook into to a python script, the `2to3` module to transform python 2 to python 3, and the python `ast` module to parse and extract method invocation calls. We note that there may be some issues in our extraction; for example, `.append` is both a python built in list method as well as a pandas method. However, we expect our trends to largely hold.

We will focus on three questions to investigate how people work with pandas.

What are some high-density functions used in interactive analysis? We studied the *total* occurrence of each pandas dataframe function in our data. The most notable ones are those used to inspect the data (`plot`, `shape`, `head`), perform numeric aggregation (`mean`, `sum`), and perform relational operations (`groupby`, `join`). It is worth noting that the notebooks contained a lot of data modification operations (both point queries as well as column and row queries) using `loc`, `iloc`, `drop`, `append`. Columns and index metadata inspection and manipulation are common as well with `index`, `columns`.

What kinds of functions are common in day-to-day usage? We counted the number of files that each pandas function has occurred in. The occurrence measures the frequency of usage per analytic job. The most commonly used functions are those that create dataframes (`read_csv`, `DataFrame`), inspect partial results (`head`, `shape`), visualizing result (`plot`), perform aggregation (`mean`, `sum`, `max`), perform point queries (`loc`, `ilo`, `ix`), add or remove data (`append`, `drop`), apply arbitrary user defined transformations (`apply`), and perform relational operations (`groupby`, `join`). It is worth noting that type-casting (`astype`) and direct access or manipulation of

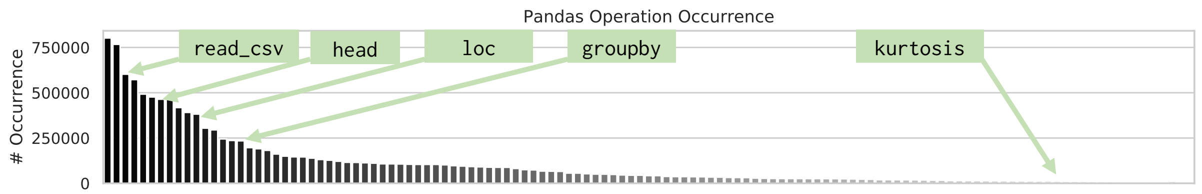


Figure 7: Pandas user statistics from GitHub dataset.

columns and index metadata, and underlying data storage (`columns`, `index`, `values`) are also high in the list.

Which functions are common used together? We also investigated the number of co-occurrence of functions in the same line of code. This typically involves the user chaining pandas functions together or calling them inside a single statement. For example, `df.dropna().describe()` is fairly common across our sample. It is also common for pandas users to perform multiple operations in single execution cell. For example, `print(result = df["col1"].mean(), df["col1"].max())` prints a tuple of summary statistics. The popularity of chained or parallel invocation suggests opportunities for acceleration, going beyond one operation at a time to more complex queries.

5. DATA MODEL CHALLENGES

Supporting the dataframe data model and algebra from Section 4 efficiently motivates a new set of research challenges. We organize these challenges based on unique properties of dataframes, and discuss their impact on query optimization, data layout, and metadata management. We first discuss the impact of flexible schemas.

5.1 Flexible Schemas, Dynamic Typing

Major challenges arise from the flexible nature of dataframe schemas. Dataframes require more than data; as noted in Section 4.2 they also require a schema to interpret the data. In the absence of explicit types for certain columns, we must run the type induction function S, and the resulting parsing functions—both of which can be expensive. Note that the type of a full column is required before we can parse the value of any cell in that column. Hence a major challenge for dataframes is to mitigate the costs inherent in flexible schemas and dynamic types.

In database terms, dataframes are more like views than tables. Programming languages like Python and R do not store data; they access data from external storage like files or databases. Hence every time a program is executed, it constructs dataframe objects anew. Unfortunately, external storage in data science is often untyped. Dataframe-friendly file formats like Apache Feather include explicit schemas and pre-parsed data, but most data files used in data science today (notably those in the ever-popular csv format) do not.

Another source of dynamism arises from schema mutations, e.g., adding or removing columns. These are first-class citizens of the dataframe algebra, unlike in relational databases, which relegate such operations to a separate DDL. As such, they are not only allowed, but are, in fact, frequent during data exploration with dataframes, especially during data preparation and feature engineering. We consider the challenge of efficient schema induction from three angles: rewriting, materialization, and query processing.

5.1.1 Rewrite Rules for Schema Induction

Due to their flexible schemas, dataframes support addition and removal of columns as first-class operations, and at any point in time could have several columns with unknown type. Certain dataframe operators need type information, however—e.g. avoid attempting to JOIN two dataframes on columns with mismatched types or using a numeric predicate on a column with some strings. The schema induction function, S, could be used to induce the requisite typing information, but it is expensive, and must be explicitly considered when modeling cost for query plans. Specifically, if certain columns are not operated on, inferring their type via S can be deferred to when they are first manipulated, and omitted entirely, if, for example, they are dropped before ever being accessed.

At least in some cases, schema inference rules might be able to avoid the application of S altogether. As one example, if ordered relational operations are chained together, schema induction can be omitted between operations, suggesting the possibility of employing rewrite rules to skip applying S. Another example involves UDFs with known output types (e.g., a MAP with a UDF that always returns an integer).

In the case of operations which merely shuffle rows around (e.g. moving even-indexed rows to the beginning of a dataframe, reordering), schema induction can be omitted entirely. When filtering or taking a sample of a dataframe, schema induction can be omitted if the type is already fairly constrained and will not be additionally constrained based on the sample. For example, if we drop all rows with strings in a specific column, we may end up with that column having a restricted type such as float or int, requiring special care.

While omitting or deferring schema inference is promising, additional complications arise from the fact that, in a dataframe system, metadata *is* data (see also Section 5.2) that may itself be queried by a user. In particular, it is common for users to perform *runtime type inspections* as a sanity check. As a result, the extra effort for eschewing or deferring schema induction may prove futile if the user chooses to inspect types anyway.

5.1.2 Reusing Type Information

It is common to reuse a dataframe across multiple statements in a program. In cases where the dataframe lacks explicit types, it can be very helpful to materialize the results of both schema induction and parsing—both within the invocation of a program (internal state), and across invocations in storage.

Materialization of flexibly-typed schemas introduces a new set of challenges. Both schema induction and parsing can be a significant fraction of the cost of processing. This raises optimization choices for materialization: we can cache the results of S (for one or more columns), and additionally we can cache the results of parsing functions (in principle, at a granularity down to the cell level). For complex multistep dataframe expressions, we can choose to make these decisions at each operator in the pipeline that introduces a dynamically-typed column. Hence the optimization search space is large. Moreover, the workload of "queries" is different from traditional materialized view settings—languages like Python are more difficult to analyze statically than SQL, and we can expect usage patterns to differ from databases as well (Section 6).

In some cases, it is reasonable to expect that a programmer will want to declare the types of the dataframe explicitly—e.g., an expression like `df_t = TRANSPOSE(df, [myschema])` where `myschema` is an array of type names for the columns. In this case, there is no need to run schema induction. In a loosely-typed language like Python, `myschema` can be an arbitrary expression returning an array of strings. For example, it might read a list of type names from a very large file with the same number of rows as `TRANSPOSE(df)`. Alternatively, the dataframe `df` itself might have "row types" stored as strings in the i'th column of the data, leading to an expression like `df_t = TRANSPOSE(df, df[i])`.

View maintenance has a role in the dataframe context, with new challenges for type induction. The most direct use is in delta-computation of expressions that have the effect of "adding" rows to their inputs. For example, consider a MAP operator with a data validation function: for each column it returns the input if it passes a validation test, else it returns an error message in that column. The new rows may all respect the constraints of the types of the input dataframe, or some new rows could break those constraints—e.g. a string-typed error message appearing in a column of numbers. In both cases, we'd like the type induction to take advantage of the work done to induce a schema for the input, and differentially decide on a schema for the output. Note that these issues get more subtle as the type system gets richer—e.g., consider an input with a column of type *percent* that is passed into an arithmetic MAP function—the output may be statically guaranteed to be numeric, and for a given dataframe may or may not still be of type *percent*.

Regardless of the source of the schema—whether it be induced, stored externally, or stored within the data itself—any implementation should assume that the metadata for a dataframe could be expensive to compute, and potentially very large. Storage and computation of this metadata can have significant overheads, and methods for ameliorating those costs will be central to scalable dataframe research.

5.1.3 Pipelining Schema Induction in Query Plans

When applying S and the parsing function to columns is unavoidable, we may be able to reduce its cost by trying to fuse it with other operations that are type-agnostic and lightweight (e.g., data movement or serialization/deserialization) while adding minimal overhead, the development of which we foresee to be a fruitful research direction.

For other operations, the position of S within the query plan can have major performance implications. Consider a MAP operation that is being applied to a column of strings. If the MAP operation is relatively inexpensive (e.g., if it is measuring the string length), it may make sense to to skip type checking via schema induction before the MAP operation. Although a type error (due to, e.g., the presence of an unexpected integer value) leads to wasted effort, it may be acceptable, as the overhead paid by actual application of the MAP is not too high. On the other hand, a MAP which performs heavy-duty regular expression parsing over long strings may delay error detection unacceptably if schema induction is fused with the MAP application.

Overall, the positioning of the schema induction operator within the query plan, by possibly fusing it with existing operators, combined with schema induction avoidance and reuse as previously discussed, is crucial for the development of a full-fledged dataframe query optimizer.

5.2 Order and Equivalence

Unlike relations, dataframes are ordered along both rows and columns—and users rely on this ordering for debugging and validation as they compose dataframe queries incrementally. This order is maintained as rows are transformed into columns and columns into rows via TRANSPOSE, ensuring near-equivalence of rows and columns. Additionally, as we saw in Section 4.4, row and column label metadata is tightly coupled with the dataframe content, and inherits the order and typing properties. In this section, we discuss the challenges imposed by enforcing order and the frequently changing schema across row and column labels and row/column orientation.

5.2.1 Order is Central

The order of a dataframe is determined by the order of ingested data. For example, a CSV file ingested as a dataframe would have the same row and column order as the file. This ordering is crucial for the trial-and-error-based interaction between a user and a dataframe system. Users expect to see the rows in their dataframe stay in the same order as they process it—allowing them to validate and debug each step by comparing its result to the previous step. For example, to ensure that a CSV file is ingested and parsed correctly, users will expect the first few rows of the dataframe to be the same as those they would see when examining the CSV file. To examine a dataframe, users will either use the operator `head/tail` to see the prefix/suffix or simply type the name of the dataframe for both the prefix and suffix in the expected order. Additionally, operators such as WINDOW and MAP from Section 4.3 expect a specific order for the rows (WINDOW) and columns (MAP). since the UDF argument to these operators may rely on that order. Dataframes also support SELECTION and PROJECTION based on the position of the rows and columns respectively.

To support order, current dataframe systems such as pandas physically store the dataframe in the same conceptual order as defined by the user. Said differently, they do not embrace physical data independence. Physical independence may open up new optimization opportunities, recognizing that as long as the displayed results preserve the desired order semantics to the users, it is not necessary that all intermediate products or artifacts (unobserved by user) adhere to the order constraint. For example, a sort operation can be "conceptual" in that a new order can be defined without actually performing the expensive sorting operation. Likewise, a transpose doesn't require the data to be reoriented in physical storage unless beneficial for subsequent operations; the transpose can be captured logically to reflect the new orientation of the dataframe.

To ensure correct semantics while respecting physical data independence, we must devise means to capture ordering information, either tracked as a separate "order column", if it is not implied via existing columns, or recording as metadata that the dataframe must be ordered based on one or more of the preexisting columns. Then, the ORDER BY on this "order column" or one of the existing columns will be treated as an operator in the query plan, and will only need to be done "on-demand" when the user requests to view a result. Additionally, since users are only ever looking at the first and/or last few lines of the dataframe, those are the only lines that are required to be ordered; we discuss this further in Section 6.1.

Extending physical data independence even further, we can adapt other data representation techniques from the database community, optimized for dataframes. This includes columnar or row-column hybrid storage [16], as well as those from scientific computing [29], array databases [69] or spreadsheets [24]. Since dataframes are neither relations, matrices, arrays, or spreadsheets, none of these representations are a perfect fit. Given that rows and columns are equivalent, one candidate for dataframe representation is as a collection of key-value pairs, where the key corresponds to the (row number, column number) pair. This representation is especially effective when the dataframe is "sparse", allowing us to omit pairs where the value is null. Then, TRANSPOSE conceptually swaps the row and column for each value: $(column, row, value)$, and can be recorded in metadata. However, some operations become more expensive, e.g. reconstructing a row for a MAP operation requires a join. Automatically detecting and updating to the right representation over the course of dataframe query execution will be a substantial challenge.

Given a certain physical representation, operations on dataframes, from a relational perspective, often make use of ordered access, e.g., editing the i^{th} row, as well as access based on the row labels, e.g., filtering based on row labels (named notation) or row position (positional notation). Because selecting the i^{th} physical row or projecting the j^{th} physical column will not necessarily correspond to selecting (resp. projecting) the desired *logical* row (resp. column), additional metadata that serves as the "order column" or "order row" must be maintained to facilitate order-independence of the physical data. Automatically maintaining indexes for this purpose can be beneficial. Recent work has developed positional indexing [25], allowing ordered access to be supported in $O(\log n)$, in the presence of edits (e.g., adding or removing rows). Column stores take a different approach to avoid expensive edits across columns, instead recording edits separately as deltas, and periodically merging them back in [16]; it would be interesting to investigate which approach is more effective for a given set of dataframe operations. Similarly, for matrices, accesses often happen in a row-major or column-major order, and identifying the right indexes to efficiently support them in conjunction with relational-style accesses, is an important challenge. In particular, when a dataframe has many rows and many columns, we may need both row- and column-oriented indexing.

5.2.2 Row/Column Equivalence

The presence of a TRANSPOSE operator in the dataframe algebra presents novel challenges in data layout and query optimization. TRANSPOSE allows users to flexibly alter their data into a desired shape or schema that can be parsed according to an appropriate schema, and queried using ordered relational operators.

To keep our data model and algebra compact, we have schemas only for columns, and our operators are defined on ordered sets of rows. By contrast, in pandas and other dataframe implementations, it is possible to perform many operations along either the rows or columns via the `axis` argument. Hence programs written in (or translated to) our algebra are likely to have more uses of TRANSPOSE than dataframe programs in the wild, to represent columnwise operations and/or to reason about per-row schemas. These operations are expressible logically in our simpler algebra by first performing a TRANSPOSE, applying the operation, and then a TRANSPOSE again to return to the original orientation. Doing frequent physical reorganizations for these operations would be a mistake, however.

The prevalence of TRANSPOSE in dataframe programs overturns many axis-specific assumptions made in traditional database storage. Axis-specific data layouts like columnar compression are problematic in this context. Metadata management also requires rethinking, since dataframes are as likely to be extremely "wide" (columnwise) as they are "tall" (rowwise). Both traditional and embedded RDBMSs typically limit the number of columns in a relation (e.g., SQL Server has an upper limit of 1024 columns, or 30k columns using the wide-table feature) [41, 62]. By applying TRANSPOSE on a tall and narrow dataframe, the number of columns can easily exceed the millions in the resulting short and wide dataframe.

Dataframe systems will need careful consideration to ensure that a TRANSPOSE call does not break assumptions made by the data layout layer. Given that the cost of performing a physical transpose will often be high, one potential way to handle the data layout layer optimization problem is to do a *logical* TRANSPOSE "pull-up". This would delay or eliminate transpose in the physical plan as much as possible since it will often destroy or render moot many existing data layout optimizations.

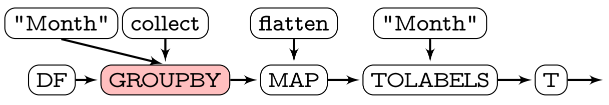
In certain cases, we may indeed want to consider optimizing the *physical* layout of the data given a TRANSPOSE operator as a part of a query plan. This is in contrast with existing data systems that create and optimize for a static data layout. A physical transpose may help the optimizer match the layout to the access pattern (e.g., matrix multiplication). A fixed data layout is likely to have a significant performance penalty when the access pattern changes. Additionally, consider a case where TRANSPOSE allows us more flexibility in query planning. In the `pivot` case in Section 4.4, we observed that transposing the result of a pivot is effectively a pivot across the other column. Specifically, if we must pivot into the wide table with `Months` as columns, we can either use the original plan (Figure 8a) or one where we proceed as if the pivot is over `Year`, but then transpose the final result so that the `Month` attribute values are used as column headers (Figure 8b). The latter plan will be faster if the optimizer leverages knowledge about the sorted order of the `Year` column to avoid hashing the groups. This is an interesting example of a new class of potential optimizations within dataframe query plans that exploit an efficient TRANSPOSE. Because the axis transpositions are happening in query expressions, the data layout becomes a physical plan property akin to "interesting orders" [70] or "hash teams" [37], expanding the rules for query optimization.

5.2.3 Metadata is Data (and Data is Metadata)

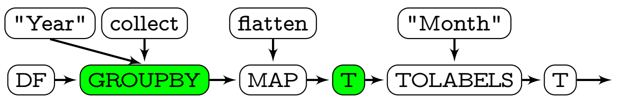
A standard feature of dataframes is the ability to fluidly move values from data to metadata and back. This is made explicit in the TOLABELS and FROMLABELS operators of our algebra, especially in combination with TRANSPOSE. These semantics cannot be represented in languages like SQL or relational algebra that are grounded in first-order logic; this is a signature of second-order logic, as explored in languages like OQL [20], SchemaSQL [49] and XQuery [26]. There is significant prior work on optimizing second-order operations like the unnesting of nested data (e.g. [33, 72, 77]). A distinguishing aspect of our setting is that a dataframe operation like TOLABELS commonly generates a volume of metadata that is dependent on the size of the data; this raises new challenges. The closest prior work to our needs is on implementing spreadsheet-style pivot/unpivot in databases (e.g., [30, 79]); this work needs to be generalized to the richer semantics of a dataframe algebra.

To address representational aspects, we could treat row labels the way we treat primary keys in a relational database—by noting the sequence of label columns in a metadata catalog. Some additional details arise in the support of positional notation: invoking TOLABELS(c_1, ..., c_n) removes the relevant columns from their positions, requiring a recalculation of the positions of all labels to the right of c_1. This can be handled by representing column order in dynamic ranked data structures like ranked B-trees [48] or range min-max trees [55]. In terms of data access, we may want to efficiently process data columns without paying to access (dynamically reassigned) metadata columns, and vice versa. In this case, columnar layouts become attractive for projection. Alternatively, labels can be moved into separate *property tables* [30], a form of "vertical partitioning" that does not rely on columnar storage layouts.

Challenges arise in more complex expressions that include both TOLABELS and other operators–notably MAP and TRANSPOSE. In these cases, the number and types of *columns* in the dataframe is data-dependent. This exacerbates the metadata storage issues discussed in the previous section, and brings up additional challenges.



(a) Original plan



(b) Optimized rewrite that leverages sorted `Year` column

Figure 8: Alternative query plans for pivoting a dataframe around the "Month" column using the algebra from Section 4.3. TRANSPOSE is abbreviated as T.

In terms of query optimization, we now have a two-dimensional estimation problem: both cardinality estimation (#rows) and *arity estimation* (#columns). For most operations in our algebra this would appear straightforward: even for TRANSPOSE, we know the cardinality and arity of output based on input. The challenge that arises is easy to see in a standard data science "macro", namely 1-hot encoding (`get_dummies` in pandas). This operation takes a single column as input, and produces a result table whose schema concatenates the input schema with an (typically large) array of boolean-typed columns, one column per distinct data value of the input. Pivot presents a similar challenge: the width of the output schema is based on the number of distinct data values in the input columns. In our algebra, these macros can be implemented using GROUPBY followed by MAP and TRANSPOSE. The resulting arity estimation problem reduces to distinct value estimation for the input to GROUPBY. Techniques like hyperloglog sketches [34] come to mind to assist here. But note that we need to compute these estimates not only on base tables that may be pre-sketched, but on intermediate results of expressions! In short, we need to do distinct value estimation for the *outputs* of query operators—including arithmetic calculations (e.g. sums, products) and string manipulations (e.g. expanding a document into constituent words).

In some scenarios, arity *estimation* is insufficient—we need exact numbers and labels of columns. Consider the example of performing a UNION of feature vectors generated from two different text corpora, say wikipedia articles unioned with DBLP articles. Each text corpus begins as a dataframe with schema (`documentID`, `content`). After a standard series of text featurization steps (word extraction with stemming and stop-word filtering followed by 1-hot encoding), each corpus becomes a dataframe with a `documentID` column, and one boolean column for each word in the corpus. The problem is that the UNION needs to dynamically check for compatibility of the input schemas—it needs to first generate the full (large!) schema for each input, and compare the two. Even if we relax our semantics to an "outer" union, we want to identify and align the common words across the corpora. These metadata requirements seem to require two passes of the inner expression's data: one to compute and align metadata, and another to produce a result. There are opportunities for optimization here to return to single-pass pipelining techniques, but they merit thoughtful investigation. This pipeline-breaking problem generalizes to any operator that reasons about its input schema(s), so it needs to be handled comprehensively. In short, we expect that the fluid movement of large volumes of data into metadata and vice versa introduces new challenges for query processing and optimization in dataframes.

6. USER MODEL CHALLENGES

Unlike in SQL where queries are submitted *all-or-nothing*, dataframe users construct queries in an incremental, iterative, and interactive fashion. Queries are submitted as a series of *statements* (as we saw in Figure 1), a few operators at a time, in trial-and-error-based *sessions*. Users rely on immediate feedback to debug and rapidly iterate on these statements and frequently revisit results of intermediate statements for experimentation and composition during exploration. This interactive session-based programming model for dataframes creates novel challenges for overall system performance and imposes additional constraints on query optimization. For example, operator reordering is often not beneficial when the results are materialized for viewing after every statement. At the same time, dataframe query development sessions are bursty, with ample think time between issuance of statements, and tolerant of incomplete results as feedback—as long as the original goals of experimentation and debugging are met, offering new opportunities for query optimization. In this section, we discuss new challenges and opportunities in query optimization arising from the interactive and incremental trial-and-error query construction of a typical user.

6.1 Interactive Feedback and Control

Dataframes are typically used in exploratory workloads, where interactive response times are crucial to providing a fluid user experience. Past studies have shown that latency in response times of greater than 500ms can lead to fewer hypotheses explored and insights generated during data exploration [51]. As another example, for data preparation—often performed on dataframes—users often rely on system feedback to guide and decide what operations to determine their next steps [39]. This feedback usually comes in the form of a display output by the dataframe system that contains a prefix or suffix of rows and columns, as in Figure 1. The need for frequent materialization of intermediate results of statements to provide feedback to the user makes it particularly difficult to satisfy the 500ms query latency requirement for interactivity. Fortunately, we can leverage two user behavior characteristics to improve interactivity: that users spend time thinking between steps, and that the inspected intermediate results are typically restricted to a prefix/suffix of rows/columns is sufficient for debugging and validation.

6.1.1 Intermediate Result Inspection & Think Time

Present-day dataframe systems such as pandas are targeted toward ensuring users can inspect intermediate results for debugging and validation, so they operate in an *eager* mode where every statement is evaluated as soon as it is issued. Program control is not returned to the user until the statement has been completely evaluated, forcing the user to be idle during that time. However, there are many cases where users do not inspect the intermediate results, or where results are discarded; in such cases, the user is still forced to wait for each statement to be evaluated. Moreover, users are either rewarded or punished based on the efficiency of a query as it is written.

On the other hand, with the *lazy* mode of evaluation, which is adopted by some dataframe-like systems [21, 31] (See Section 7), control is returned to the user immediately, and the system defers the computation until the user requests the result. By scheduling computation later, the system can wait for larger query sub-expressions to be assembled, leading to greater opportunities for optimization. The downside of lazy evaluation is that computation only begins when the user requests the result of a query. This introduces new burdens for users, particularly for debugging, since bugs are not revealed until computation is triggered.

For example, consider two commutative operations `op1`, and `op2`. Say the user submits the statement `x = df.op1()` followed by `y = x.op2()`. In eager evaluation, `x` will be fully materialized before execution begins on `y`, even if `x` is never used again. Computing `y` could be done using `df.op2().op1()`, but it is often more beneficial to use the materialized version of `x` instead. In lazy evaluation, execution will be deferred until explicitly requested, so the expression that creates `y` could be optimized to run `df.op2().op1()`. The downside of this approach is that the user has to wait until they explicitly request `y` before they realize that there is a potential bug in `x`.

Furthermore, neither the lazy nor the eager mode take advantage of the fact that the users spend time thinking between steps: the system is idle during think-time. We can can leverage this time for computation, allowing us to effectively achieve the benefits of both paradigms. While interactive latency is important to support immediate feedback, recent empirical studies have also shown that optimizations can be relaxed to account for user's long think time between operations in exploratory analysis [22]. We describe a novel *opportunistic* query evaluation paradigm suitable for optimizing dataframes in an interactive setting.

Like lazy evaluation, opportunistic evaluation does not require the user to wait after each statement. Instead, the system opportunistically starts execution, while passing control back to users with a pointer to the eventually computed dataframe (a "future"), which is asynchronously computed in the background. We can then use system resources to compute results in the background as users are composing the next step. Like eager evaluation, opportunistic evaluation does not wait for users to complete the entire query to begin evaluation. However, when a user requests to view a certain output, opportunistic evaluation can prioritize producing that output over all else. Opportunistic evaluation allows queries to be rewritten as new statements are submitted (e.g., `df.op2().op1()`) to get to the requested answer as fast as possible, taking into account what is partially computed. There are also new opportunities within opportunistic evaluation to do speculation, where during idle time the system can start executing statements that commonly follow previous ones. Opportunistic evaluation also leads to new challenges in sharing and reuse across many query fragments whose computation has been scheduled in the background (see also Section 6.2).

6.1.2 Prefix and Suffix Inspection

The most common form of feedback provided by dataframe systems is the tabular view of the dataframe, as shown in Figure 1. The tabular view serves as a form of visualization that not only allows users to inspect individual data values, but also convey the structural information associated with the dataframe. Structural information, especially as it relates to order, is important for validating the results of queries that manipulate and reshapes the dataframe. This tabular visualization typically contains a partial view of the dataframe, displaying the first and last few rows of the dataframe, accessed using `head`, `tail`, or other print commands.

One way to give the users immediate feedback is to return the output to the user as soon as these k rows are assembled, computing the rest of the output in the background using opportunistic evaluation. This is reminiscent of techniques that optimize for early results [75, 76] for LIMIT queries [47], or for representative tuple identification [73], but a key difference in dataframes is that order must be preserved (so "any-k" result tuples will not suffice [47]), and there are many more blocking operators. One starting point would be to design or select physical operator implementations that not just prioritize high output rate [75], but also preserve order, thereby ensuring that the first k rows will be produced as quickly as possible. As an example, if only the first k rows of an ordered join were to be computed, a nested loop join, with the result displayed after k rows are computed, might work well. We can progressively process more portions of the input dataframes until k output rows are produced in order: this may mean processing more than k rows of the inputs if there are very selective predicates. Figuring out the right way to exploit parallelism to prioritize processing the prefixes of the ordered input dataframes to produce the ordered prefix of the output is likely to be a substantial challenge.

Additionally, certain blocking operators will cause problems. While returning the first k rows following a TRANSPOSE, especially when using columnar storage, can be fairly efficient, it may be hard to produce the first k tuples of a GROUP BY or SORT without examining the entire data first. (Indeed, for small enough k, sorting may be faster than $O(n \log n)$, but still requires an $O(n)$ sequential scan.) SORT is an obvious example where the top k rows cannot be narrowed down a priori; a full scan is inevitable also for GROUPBY on non-clustered columns. That said, since the top and bottom k rows are often the only results inspected for dataframe queries, we may benefit from materializing additional intermediate results or supporting indexes to retrieve them efficiently, without exorbitant storage overhead. We could, for example, materialize the prefix and suffix of a dataframe in original and transposed orientations, or the prefix or suffix of the dataframe sorted by various columns to allow for efficient processing subsequently. These materializations could happen during think-time as discussed in Section 6.1.1. We may also be able to exploit approximate query processing to produce the prefix/suffix early for blocking operators [18, 32, 40, 59, 81]. Since the tabular view is only a special form of visualization, a rich body of related work from visualization on how to allow users to quickly but approximately make decisions or perform debugging or validation, may be applicable [19, 46, 52, 58]; however, the rich space of operators that goes beyond simple GROUPBY aggregation will lead to new challenges. Another interesting usability-oriented challenge is whether this tabular view of prefixes or suffixes is indeed best for debugging—perhaps highlighting possible erroneous values or outliers in dataframe rows or columns that are not in the prefix or suffix may also be valuable [63].

While it is well known that some aggregates like MAX cannot be approximated [54], even blocking operators such as SORT can return early approximate results, using results from the 1990s [65]. There may be additional opportunities for approximation if the user simply wants to inspect the approximate structure of the result for debugging purposes, especially in conjunction with prefix/suffix computation. For example, we can provide the overall structure of the output of a pivot table computation (displaying the row-wise groups and column headers), without actually filling in any of the aggregate values, and doing so progressively. Similar ideas of adding "placeholder" values for in-progress tuples have been proposed in streaming [64], web-database hybrid [36], and crowdsourcing [57] contexts, but not, as far as we can tell, for a group-by aggregation setting. This idea could also be applied, for example, to TRANSPOSE, where the structure of the output dataframe is prepared first, with the values filled in progressively. Other notions of approximation may also be valuable, e.g., the incomplete/phantom notions in Lang et al. [50], wherein the result may contain additional rows not present in the dataframe query result, or rows that should be present, but are absent. This could be valuable, for example, for expensive filters.

In fact, we could also exploit correlations [45] between the filtering attribute and the other attributes in order to quickly approximate the rows that might pass the filter and quickly display them to the user, refining as additional filter evaluations are performed.

6.2 Incremental Query Construction

In addition to the challenges around satisfying the stringent latency requirement for immediate feedback discussed above, the need to frequently evaluate and display results for intermediate sub-expressions (i.e., the results of statements) over the course of a session complicates query optimization, as we saw in Section 6.1.1. While incrementally constructing dataframe queries over the course of an interactive session, users iterate on query sub-expressions through trial-and-error, frequently inspecting and revisiting intermediate results to try alternate exploration paths. Such fragmented workloads limit the optimizations that can be applied to each subexpression. However, since user statements often build on others, we can jointly optimize across these statements and resulting subexpressions, sharing work as much as is feasible. Further, since users commonly return to old statements to try out new exploration paths, we can leverage materialization to avoid redundant reexecution. We discuss these two ideas next.

6.2.1 Composable Subexpression Support

As a result of opportunistic evaluation, there are often many statements that are not completely executed when issued by the user, and are instead executed in the background asynchronously during user think time. Moreover, by prioritizing the return of a prefix or suffix of the results (Section 6.1.2), often, many statements are not computed entirely, with the computation either deferred (in lazy or eager evaluation), or being scheduled in the background (in opportunistic evaluation). Thus, there are many statements that may be scheduled for execution at the same time. These statements may operate over similar or identical subsets of data. These overlapping queries that can be batch processed make dataframes particularly amenable to multi-query optimization (MQO), e.g., [35, 38, 67, 71]. In fact, some have argued that MQO has limited applicability in a general relational context: "One problem of MQO is its limited applicability (...). In many workloads (...) there aren't many opportunities to factor out common subexpressions" [35], and "the synchronization of the execution of queries with common subexpressions when queries are submitted at different moments in time" [35]. In the dataframe setting, both these reasons for limited applicability do not hold: there are often many statements executed essentially in sync, and there are lots of opportunities to factor out common subexpressions since these statements essentially build on top of each other. However, new challenges emerge because of the new space of operators, as well as the prioritization of the return of prefixes/suffixes over the entire result when requested by the user.

One simple approach is to allow operations that share inputs to share scans, thereby reducing the overhead required to access data. We can go even further if we recognize that many statements are essentially portions of a query composed incrementally (e.g., a `TRANSPOSE` followed by a `PROJECT` to simulate `SELECT`). Therefore, we simply need to construct a query plan wherein subplans that correspond to intermediate dataframe results are materialized as a by-product. These intermediates are also likely to be reused by the user in the future. This poses an interesting conundrum, because ensuring that the sub-plan results are materialized "along the way" may result in suboptimal overall plan selection, which is problematic when the user cares more about the final dataframe than intermediates. For example, the optimal way to compute a `SELECT` may not be to first compute a `TRANSPOSE` and then do a `PROJECT`, even though this may have the benefit of producing the appropriate intermediate results. Unlike MQO in relational databases, wherein it is important to share join subexpressions, here, an even more expensive operation is `TRANSPOSE`—necessitating sharing if at all possible. By using partial results to help users avoid debugging mistakes, we may be able to reduce the importance of constructing many of the intermediate results in entirety, unless requested by the user explicitly. Moreover, by observing the user's likelihood of inspecting the intermediates over the course of many sessions, we can do a weighted joint optimization of all query subexpressions, where the weights for each intermediate dataframe corresponds to its importance. Going one step further, we can try to jointly optimize not just the evaluation of intermediate and final result dataframes, but also the partial or approximate results—a challenging endeavor. We can estimate probabilities for what the user might do next, e.g., inspect an intermediate, or compose the next statement, and the time they may take to do so. We can couple that with quantifying the benefit of the user seeing a certain portion of an intermediate result at a certain time, to construct a globally optimal query plan.

6.2.2 Debugging & Building Queries Incrementally

The incremental and exploratory nature of dataframe query construction over the course of a session leads to nonlinear code paths wherein the users revisit the same intermediate results repeatedly as a step towards constructing just the right queries they want. In such cases, intelligently materializing key intermediate results can save significant redundant computation and speed up query processing. The optimizer needs to handle the trade off between materialization overhead and the reduced execution time facilitated by availability of such intermediates to utilize storage in a way that maximizes saved compute—small intermediate dataframes that are time-consuming to compute and reused frequently should be prioritized over large intermediate dataframes that are fast to compute. Note however that materialization doesn't necessarily need to happen on-the-fly, and can be also performed in the background asynchronously during during user think-time. Determining what to materialize requires us to predict which intermediates are likely to be used frequently. The prediction algorithm should take into consideration several factors, including user intent, past workflows, and operator lineage.

Depending on the underlying intent, users can interact with dataframes in very different ways. A user who is performing data cleaning is likely to issue point queries and focus on regions with missing or anomalous values; users exploring the data for building machine learning models tend to focus on manipulating columns with high mutual information with the target column, or more broadly on feature engineering. Taking advantage of user intent can lead to highly effective materialization and reuse strategies befitting specific access patterns, such as in machine learning workflows [80]. The interactive sessions in dataframe development make it possible for the system to infer and adapt to user intent.

User intent inference involves extensive offline analysis of workloads with known intents as well as online processing of relevant telemetry: recent Jupyter notebook corpora can provide a promising starting point [68]. One challenge is that unlike SQL workloads, dataframe queries tend to be interleaved with non-dataframe operators in the same session, which requires special considerations to identify the dataframe portion of the workload and to handle the interaction between the dataframe system and other frameworks.

Finally, dataframe queries in a session often build upon one another. In the dataflow graph of dataframe queries, we are likely to see intermediate results that lie on the path to many leaf nodes. A simple heuristic is to persist intermediate results with high fanouts; more advanced graph analysis techniques can be applied to determine prominent intermediate results. Opportunistic evaluation can significantly complicate the analysis as the execution order can differ drastically from the query order.

In terms of costing operators for materialization and reuse, the dataframe setting introduces two novel challenges. Partial views to support fast inspection in conjunction with opportunistic evaluation can break up operators into multiple partial operators evaluated at different times, motivating the need for short and long term costs on partial views for each operator. The materialization and reuse decisions derived from these costs can feed back into the decisions on filtering for partial views or delaying evaluation. For example, if several queries based on a new sort order require immediate feedback in the near future, it might be prudent to incur a delay

Feature	Modin	Pandas	R	Spark	Dask
Ordered model	✓	✓	✓	✓†	
Eager execution	✓	✓	✓+		
Row/Col Equivalency	✓	✓	✓		
Lazy Schema	✓	✓	✓		✓
Relational Operators	✓	✓	✓	✓	✓
MAP	✓	✓	✓	✓	✓
WINDOW	✓	✓	✓	✓	✓
TRANSPOSE	✓	✓	✓		
TOLABELS	✓	✓	✓		✓*
FROMLABELS	✓	✓	✓		

Table 3: Table of comparison between dataframe and dataframe-like implementations. Blue indicates dataframe systems, red indicates dataframe-like implementations. †: Spark can be treated as ordered for some operations. +: R dataframe operators can be invoked lazily or eagerly. *: Dask sorts the dataframe by the row labels after TOLABELS.

on the first query to materialize the new sort order in its entirety, in order to significantly speed up subsequent queries on the new order through reuse. Of course, being able to make such decisions hinges on the ability to predict future reuse as discussed above. Secondly, the constantly growing dataflow graph requires eviction of old materialized results from memory. The interesting challenge in the dataframe context is that future reuse is determined by both what the user will do in the future and what the opportunistic evaluator will choose to compute, with the former being purely speculative and the latter being known within the system. We can reconcile the "two futures" by passing the model we build of the future workflow to the opportunistic scheduler for unified materialization/reuse planning.

Another approach to speed up dataframe queries would be to defer the creation of new dataframes as a result of queries and instead allow for the results of dataframe queries to be essentially non-materialized "views". This could be useful, for example, when a dataframe query essentially adds a new derived column for feature engineering. In this case, we don't actually add the derived column and create a new dataframe, simply recording the operations instead, and materializing the result on-demand. Deferring the operations also opens up opportunities for pipelining through subsequent operations, saving overall computation costs. In fact, the Vaex project [15], which is a dataframe-like system (as we described in Section 4.2) that supports querying on HDF5 files, implements virtual columns. With virtual columns, the column is not actually materialized until required for output, printing, or for a query. In cases where the computation that creates a column is expensive, virtual columns will need to be paired with intelligent caching mechanisms that prioritize caching columns that were expensive to generate.

7. RELATED WORK

While our focus on pandas is driven by its popularity, in this section, we discuss other existing dataframe and dataframe-like implementations. Table 3 outlines the features of these dataframe and dataframe-like implementations. We will discuss how existing dataframe implementations fit into our framework, thus showing how our proposed research is applicable to these systems.

Data model and algebra. To the best of our knowledge, an algebra for dataframes has never been defined previously. Recent work by Hutchinson et al. [42, 43] proposes an algebra called Lara that combines linear and relational algebra, exposing only three operators: JOIN, UNION, and Ext (also known as "flatmap"); however, dataframe metadata manipulation operators are not supported. Other differences stem from the flexible data model and lazily induced schema. We will draw on Lara as we continue to refine our algebra.

Dataframe Implementations: R. As we discussed in Section 4.1, the R language (and the S language before it), both support dataframes in a manner similar to pandas and can be credited for initially popularizing the use of dataframes for data analysis [44]. R is still quite popular, especially among the statistics community. An R dataframe is a list of variables, each represented as a column, with the same number of rows. While both the rows and columns in an R dataframe have names, row names have to be unique; thus the pandas dataframe is more permissive than the R one. As shown in Table 3, R supports all of the operations in our algebra. The R dataframe fully captures our definition of a dataframe, and thus, implementational support of R dataframes requires only conforming the R API to our proposed algebra. External R packages such as readr, dplyr, and ggplot2 operate on R dataframes and provide functionalities such as data loading, transformation, and visualization, similar to ones from the pandas API [5, 78].

Dataframe-like Implementations. Some libraries provide a functional or object-oriented programming layer on top of relational algebra. These libraries include SparkSQL dataframes [2], SQL generator libraries like QueryDSL [3] and JOOQ [1], and object relational-mapping systems (ORMs) such as Ruby on Rails [4] and SQLAlchemy [23]. All of these systems share some of the benefits with respect to incremental query construction mentioned in Section 6. However, they generally do not support the richness and expressiveness of dataframes , including ordering of rows, symmetry between rows and columns, and operations such as transpose.

SparkSQL and Dask are scalable dataframe-like systems that take advantage of distributed computing to handle large datasets. However, as shown in Table 3, Spark and Dask do so at the cost of limiting the supported dataframe functionalities. For example, a dataframe in SparkSQL does not treat columns and rows equivalently and requires a predefined schema. As a consequence, SparkSQL does not support TRANSPOSE and is not well optimized for dataframes where columns substantially outnumber rows. Thus, SparkSQL is closer to a relation than a dataframe. Koalas [11], a wrapper on top of the SparkSQL API, attempts to be more dataframe-like in the API but suffers from the same limitations.

Dask enables distributed processing by partitioning along the rows and treating each partition as a separate "dataframe", thus acting as a "meta-dataframe". Since ordering and transpose are ill-defined for a group of dataframes, Dask fundamentally cannot support operations that rely on row-ordering. The set of operations supported are restricted to those that can be combined into a single output based on the resulting, constituent dataframes. These include embarassingly parallel operations, such as filter, aggregation, groupby, and join.

Unlike these systems, MODIN treats the dataframe data model and algebra as first-class citizens, as opposed to a means to enable distributed processing, addressing challenges in dataframe processing in systems like pandas and R at scale, while not sacrificing the convenient functionalities that have made dataframes so popular. We advocate that our research vision around the data model proposed in this paper is a key component towards this more holistic approach for optimizing dataframe systems.

8. CONCLUSION

In recent years, the convenience of dataframes have made them the tool of choice for data scientists to perform a variety of tasks, from data loading, cleaning, and wrangling to statistical modeling and visualization. Yet existing dataframe systems like pandas have considerable difficulty in providing interactive responses on even moderately-large datasets of less than a gigabyte. This paper outlines our research agenda for making dataframes scalable, without changing the functionality or usability that has made them so popular. Many fundamental assumptions made by relational algebra are entirely discarded in favor of new ones for dataframes, including rigid schemas, an unordered data model, rows and columns being distinct, and a compact set of operators. Informed by our experience in developing MODIN, a drop-in replacement for pandas, we described a number of research challenges that stem from revisiting familiar data management problems, such as metadata management, layout and indexing, and query planning and optimization, under these new assumptions. As part of this work, we also proposed a candidate formalism for dataframes, including a data model as well as a compact set of operators, that allowed us to ground our research directions on a firm foundation. We hope our work serves as a roadmap and a call-to-action for others in the database community to contribute to this emergent, exciting, and challenging research area of scalable dataframe systems development.

Acknowledgments

We would like to acknowledge those who have made significant contributions to the MODIN codebase: we thank Omkar Salpekar, Eran Avidan, Kunal Gosar, GitHub user ipacheco-uy, Alex Wu, and Rehan Sohail Durrani. We also thank Ion Stoica for initial discussions and encouragement.

9. REFERENCES

[1] Manual: JOOQ v3.12. https://www.jooq.org/doc/3.12/manual-single-page/. Date accessed: 2019-12-27.

[2] PySpark 2.4.4 Documentation: pyspark.sql module. http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions. Date accessed: 2019-12-27.

[3] Reference Guide: QueryDSL v4.1.3. http://www.querydsl.com/static/querydsl/4.1.3/reference/html_single/. Date accessed: 2019-12-27.

[4] Ruby on Rails. https://rubyonrails.org/. Date accessed: 2019-12-27.

[5] Tidyverse: R packages for data science. https://www.tidyverse.org/. Date accessed: 2019-12-27.

[6] A Beginner's Guide to Optimizing Pandas Code for Speed, Medium Blog. https://bit.ly/2v4ZvLQ, 2017. Date accessed: 2019-12-27.

[7] Python's Explosion Blamed on Pandas, The Register UK. https://www.theregister.co.uk/2017/09/14/python_explosion_blamed_on_pandas/, 2017. Date accessed: 2019-12-27.

[8] The Incredible Growth of Python, Stack Overflow Blog. https://stackoverflow.blog/2017/09/06/incredible-growth-python/, 2017.

[9] Why is Python Growing So Quickly? Stack Overflow Blog. https://stackoverflow.blog/2017/09/14/python-growing-quickly/, 2017. Date accessed: 2019-12-27.

[10] Enhancing performance, Pandas Documentation. https://pandas.pydata.org/pandas-docs/stable/user_guide/enhancingperf.html, 2019. Date accessed: 2019-12-27.

[11] Koalas: pandas api on apache spark. https://koalas.readthedocs.io/en/latest/, 2019. Date accessed: 2019-12-27.

[12] Minimally Sufficient Pandas. https://medium.com/dunder-data/minimally-sufficient-pandas-a8e67f2a2428, 2019. Date accessed: 2019-12-27.

[13] Pandas API reference. https://pandas.pydata.org/pandas-docs/stable/reference/index.html, 2019. Date accessed: 2019-12-27.

[14] R: Data Frames. https://stat.ethz.ch/R-manual/R-devel/library/base/html/data.frame.html, 2019. Date accessed: 2019-12-27.

[15] Vaex: Out-of-core dataframes for python. https://github.com/vaexio/vaex, 2019. Date accessed: 2019-12-27.

[16] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[17] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[18] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[19] D. Alabi and E. Wu. Pfunk-h: Approximate query processing using perceptual models. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 10. ACM, 2016.

[20] A. M. Alashqur, S. Y. Su, and H. Lam. Oql: a query language for manipulating object-oriented databases. In *Proceedings of the 15th international conference on Very large data bases*, pages 433–442. Morgan Kaufmann Publishers Inc., 1989.

[21] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[22] L. Battle and J. Heer. Characterizing Exploratory Visual Analysis: A Literature Review and Evaluation of Analytic Provenance in Tableau. *Eurographics Conference on Visualization (EuroVis) 2019*, 38(3), 2019.

[23] M. Bayer. Sqlalchemy. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org, 2012.

[24] M. Bendre, B. Sun, D. Zhang, X. Zhou, K. C.-C. Chang, and A. Parameswaran. Dataspread: Unifying databases and spreadsheets. *Proceedings of the VLDB Endowment*, 8(12):2000–2003, 2015.

[25] M. Bendre, V. Venkataraman, X. Zhou, K. Chang, and A. Parameswaran. Towards a holistic integration of spreadsheets with databases: A scalable storage engine for presentational data management. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 113–124. IEEE, 2018.

[26] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. Xquery 1.0: An xml query language. *W3C working draft*, 7, 2001.

[27] J. Chambers, T. Hastie, and D. Pregibon. Statistical models in s. In K. Momirović and V. Mildner, editors, *Compstat*, pages 317–321, Heidelberg, 1990. Physica-Verlag HD.

[28] J. M. Chambers, T. J. Hastie, et al. *Statistical models in S*, volume 251. Wadsworth & Brooks/Cole Advanced Books & Software Pacific Grove, CA, 1992.

[29] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *[Proceedings 1992] The Fourth*

Symposium on the Frontiers of Massively Parallel Computation, pages 120–127. IEEE, 1992.

[30] C. Cunningham, C. A. Galindo-Legaria, and G. Graefe. Pivot and unpivot: Optimization and execution strategies in an rdbms. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 998–1009. VLDB Endowment, 2004.

[31] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[32] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample+ seek: Approximating aggregates with distribution precision guarantee. In *Proceedings of the 2016 International Conference on Management of Data*, pages 679–694. ACM, 2016.

[33] L. Fegaras. Query unnesting in object-oriented databases. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 49–60, 1998.

[34] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. 2007.

[35] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment*, 5(6):526–537, 2012.

[36] R. Goldman and J. Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *ACM SIGMOD Record*, volume 29, pages 285–296. ACM, 2000.

[37] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in microsoft sql server. In *VLDB*, volume 98, pages 86–97. Citeseer, 1998.

[38] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 383–394. ACM, 2005.

[39] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. *CIDR 2015 - 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[40] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Acm Sigmod Record*, volume 26, pages 171–182. ACM, 1997.

[41] R. D. Hipp. Sqlite, 2020.

[42] D. Hutchison, B. Howe, and D. Suciu. Lara: A key-value algebra underlying arrays and relations. *arXiv preprint arXiv:1604.03607*, 2016.

[43] D. Hutchison, B. Howe, and D. Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, page 2. ACM, 2017.

[44] R. Ihaka and R. Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[45] M. Joglekar, H. Garcia-Molina, A. Parameswaran, and C. Re. Exploiting correlations for expensive predicate evaluation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1183–1198. ACM, 2015.

[46] A. Kim, E. Blais, A. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *Proceedings of the VLDB Endowment*, 8(5):521–532, 2015.

[47] A. Kim, L. Xu, T. Siddiqui, S. Huang, S. Madden, and A. Parameswaran. Optimally leveraging density and locality for exploratory browsing and sampling. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 7. ACM, 2018.

[48] D. E. Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[49] L. V. Lakshmanan, F. Sadri, and I. N. Subramanian. Schemasql-a language for interoperability in relational multi-database systems. In *VLDB*, volume 96, pages 239–250. Citeseer, 1996.

[50] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1275–1286. ACM, 2014.

[51] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics*, 20(12):2122–2131, 2014.

[52] S. Macke, Y. Zhang, S. Huang, and A. Parameswaran. Adaptive sampling for rapidly matching histograms. *Proceedings of the VLDB Endowment*, 11(10):1262–1275, 2018.

[53] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017.

[54] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 38(3):3–29, 2015.

[55] G. Navarro and K. Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):1–39, 2014.

[56] New York (N.Y.). Taxi And Limousine Commission. New york city taxi trip data, 2009-2018, 2019.

[57] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1203–1212. ACM, 2012.

[58] Y. Park, M. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 755–766. IEEE, 2016.

[59] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476. ACM, 2018.

[60] F. Perez and B. E. Granger. Project jupyter: Computational narratives as the engine of collaborative data science. *Retrieved September*, 11(207):108, 2015.

[61] D. Petersohn, W. Ma, D. Lee, S. Macke, D. Xin, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran. Towards scalable dataframe systems. *arXiv preprint arXiv:2001.00888*, 2020.

[62] M. Raasveldt and H. Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019.

[63] V. Raman and J. M. Hellerstein. Potter ' s Wheel : An Interactive Data Cleaning System. *Proceedings of the 27th VLDB Conference*, 2001.

[64] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 275–286. ACM, 2002.

[65] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB*, volume 99, pages 709–720, 1999.

[66] L. A. Rowe and M. R. Stonebraker. The postgres data model. *Readings in object-oriented database systems*, pages 461–473, 1990.

[67] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.

[68] A. Rule, A. Tabard, and J. D. Hollan. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 32:1–32:12, New York, NY, USA, 2018. ACM.

[69] F. Rusu and Y. Cheng. A survey on array storage, query languages, and systems. *arXiv preprint arXiv:1302.0103*, 2013.

[70] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[71] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[72] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying xml views of relational data. In *VLDB*, volume 1, pages 261–270, 2001.

[73] M. Singh, A. Nandi, and H. Jagadish. Skimmer: rapid scrolling of relational query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 181–192. ACM, 2012.

[74] J. W. Tukey. *Exploratory data analysis*, volume 2. Reading, Mass., 1977.

[75] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 37–48. ACM, 2002.

[76] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 285–296. VLDB Endowment, 2003.

[77] S. Wang, E. A. Rundensteiner, and M. Mani. Optimization of nested xquery expressions with orderby clauses. *Data & Knowledge Engineering*, 60(2):303–325, 2007.

[78] H. Wickham. Tidy data. *The Journal of Statistical Software*, 59, 2014.

[79] C. M. Wyss and E. L. Robertson. A formal characterization of pivot/unpivot. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 602–608, 2005.

[80] D. Xin, S. Macke, L. Ma, J. Liu, S. Song, and A. Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *Proceedings of the VLDB Endowment*, 12(4):446–460, 2018.

[81] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 277–288. ACM, 2014.