Dataframe Systems: Theory, Architecture, and Implementation

Devin Petersohn

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-193
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-193.html

August 13, 2021

Acknowledgement

This work would not have been possible without the support of many people.

I would like first to thank Professor Anthony Joseph, Professor Aditya Parameswaran, and Professor Fernando Perez who all helped me contextualize my work and gave excellent feedback on how to improve this work.

There were many collaborators who put a significant effort into the work presented here. Joe Hellerstein, Joey Gonzalez, and Ion Stoica were all instrumental in getting this work off the ground and contributing significant, meaningful ideas and work around dataframes.

Finally, I would like to thank my wife Madelyn and daughter Vivian. Their support through the difficult parts of the PhD have been critical to my success.

Dataframe Systems: Theory, Architecture, and Implementation

by

Devin Petersohn

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Anthony Joseph, Chair
Assistant Professor Aditya Parameswaran
Assistant Professor Fernando Pérez

Summer 2021

Dataframe Systems: Theory, Architecture, and Implementation

Abstract

Dataframe Systems: Theory, Architecture, and Implementation

by

Devin Petersohn

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Anthony Joseph, Chair

Dataframes are a popular abstraction to represent, prepare, and analyze data. Despite the remarkable success of dataframe libraries in R and Python, dataframe operations face performance issues even on moderately large datasets. Moreover, there is significant ambiguity regarding dataframe semantics. In this thesis, 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. While most modern systems aim to scale dataframe workloads by changing properties of dataframes – or by requiring users to be proficient at distributed systems – we instead target supporting scalable dataframe operations without changing their semantics. This dissertation takes a ground-up approach towards scaling dataframe systems, starting with a formal dataframe data model and algebra, and ending with a reference implementation. This implementation, Modin, has already accumulated significant community support: over 6,000 GitHub stars and 1 million installs to date. This interest shows the need for systems that solve modern data science problems without changing semantics. Included in this thesis are several of our insights into how to build systems for data scientists and what aspects data scientists prioritize. We believe these insights were instrumental in unlocking the interest and support from the community in our open source work.

To Madelyn and Vivian

This work is dedicated to the ones who believed in me before anyone else. We did it.

Contents

List of Figures

List of Tables

Acknowledgments

This work would not have been possible without the support of many people.

I would like first to thank Professor Anthony Joseph for advising me through the ups and downs of graduate school, and for all of the technical and personal advice he gave over the course of my graduate school career. Without this advice I certainly would not have made it to the end.

I would like to thank Professor Aditya Parameswaran for the deep collaboration and for serving on my committee. Aditya's collaboration and paper-writing help was instrumental in helping me get to this point.

Additionally, I would like to thank my committee member Professor Fernando Perez who helped me contextualize my work in the scientific Python community, and gave excellent feedback on how to integrate with that community.

There were many collaborators who put a significant effort into the work presented here. Joe Hellerstein, Joey Gonzalez, and Ion Stoica were all instrumental in getting this work off the ground and contributing significant, meaningful ideas and work around dataframes.

I would also like to thank Areg Melik-Adamyan and the group at Intel for their help during my PhD. Areg supported the development of Modin and my PhD, and collaborating and working with Areg helped make Modin successful.

I would like to acknowledge all of the contributors and committers to Modin, past and present. Their contributions have helped the open source project grow to be widely adopted, and I could not have done that alone.

Finally, I would like to thank my wife Madelyn and daughter Vivian. Their support through the difficult parts of the PhD have been critical to my success.

Chapter 1

Introduction

Data science has become one of the fastest growing and most cross-cutting fields with the advent of ubiquitous systems for data-centric computation [49]. Tools such as the R programming language [113] and pandas library [69] for Python have created accessible interfaces for manipulating data and performing exploratory data analysis (EDA) [118], the process of summarizing, understanding, and deriving value from a dataset. These tools have enabled organizations, both large and small, to better extract insight from collected or generated data. Further, the ability for such tools to run on commodity, highly available hardware, with open-source implementations and community support, has pushed even organizations without technical roots in mathematics or computer science to seek out data science solutions. Concurrently, the rate of data creation and collection has been accelerating over the past decade with no indication of a slow-down. This growth causes problems for traditional systems which are constrained to single-node, single-threaded implementations, despite the number of cores on an individual machine constantly increasing. This increase in the rate of data generation and collection has spurred advances in distributed and parallel computation, with many frameworks embracing computing across a cluster for data processing and analysis [33, 9, 116, 78].

However, data scientists, who may have training in mathematics or statistics rather than computer science, often do not possess deep computational systems expertise. As such, they may have inferential reasoning skills and the knowledge to manipulate or draw insights from the data, but may be unfamiliar with best practices for engineering or adapting fast, scalable software to work with said data. As the average data scientist is familiar with abstractions such as pandas or R to work with their data, it is unreasonable for a data scientist to work directly with low level libraries such as MPI [37] or OpenMP [29]. Even interfaces that computer scientists label as "high level", such as MapReduce [33], SQL, or Spark DataFrames [109] may be out of reach for a typical data scientist. Data scientists typically interact with their data in interactive environments using ad-hoc methods. For example,

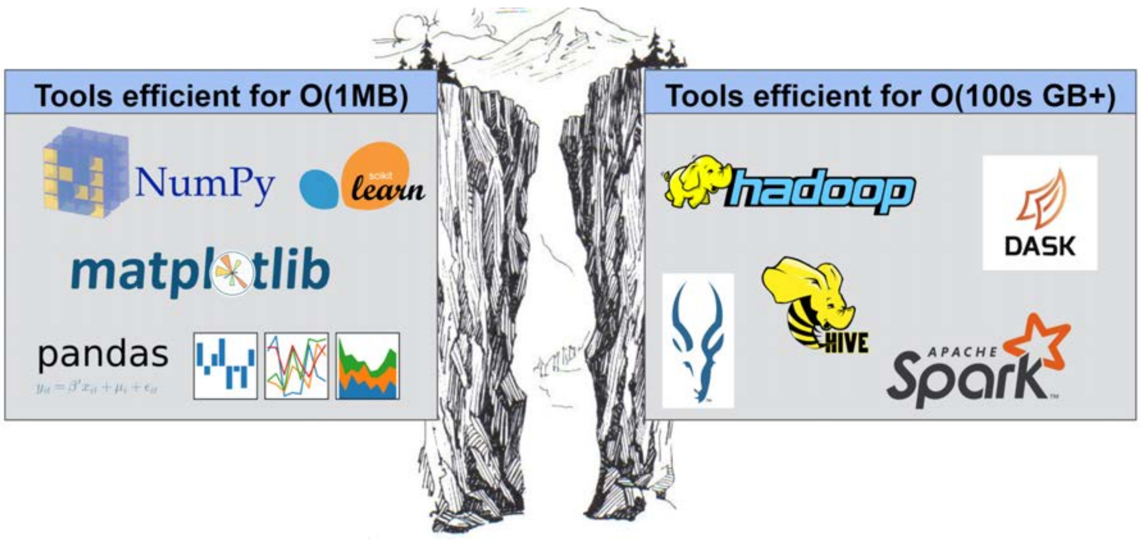


Figure 1.1: The data dcience landscape at small and large scale

in Python, a typical data science workflow might start with the user initially inspecting the quality of data by loading the dataset into pandas, performing a variety of cleaning methods based on the user's manual inspection of the data and summary statistics, as well as generation of insights via various grouping, summarization, and statistical commands, all in an interactive environment such as IPython [86] or Jupyter Notebook [61]. The actions the users perform on their data are usually dependent on the results they receive on previous views of said data.

This ad-hoc approach towards handling data is disjoint from the approach taken by many popular parallel computation libraries, such as Dask [96], Spark [9], and Tensorflow [2], that design computation around building and lazily executing entire dataflow graphs. The advantage to the latter designs is the ability to perform holistic query optimization on the entire graph, providing more performance gains than eagerly evaluated frameworks. However, data scientists are accustomed to eager evaluation-based tools, like pandas, which have more natural interoperability with interactive environments such as Jupyter notebooks.

As data scientists move from smaller to larger datasets, they regularly encounter a change in requirements, user interface, and user experience, which can be extremely frustrating. Figure 1.1 illustrates this disconnect. On the left, there are tools that are designed for single-node usage. These are the tools that are used to teach students how to do data science, and they are tools that every data scientist knows well. On the left is a sample of the tools that are used in "large scale" data science. These tools solve scalability problems well, but do not expose the same API as those tools on the left. Perhaps more importantly, these tools typically require some distributed computing knowledge, e.g., the number of partitions to choose for your data. Our goal in this dissertation is to start bridging the divide between the tools on the right and on the left of Figure 1.1. We aim to preserve the ease of use of the tools on the left, while providing the scalability benefits of the tools on the right.

For concreteness, we focus on the pandas dataframe library due to its ubiquity [70]. Dataframes are a popular abstraction to represent, prepare, and analyze data. However, despite the remarkable success of pandas and similar dataframe libraries in R, dataframes face performance issues even on moderately large datasets. Moreover, there is significant ambiguity regarding dataframe semantics. In this dissertation we lay out a vision and roadmap for scalable dataframe systems. We propose an evolution of existing dataframe libraries such as pandas, based on solid theoretical foundations, that can scale to modern data volumes without sacrificing semantics. 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 accumulated more than 25,000 stars on GitHub. Python's own popularity has been attributed to the success of pandas for data exploration and data science [124, 89].

In Chapter 2, we take a deep dive into the nuances of dataframe user behavior. We explore in detail how this behavior might impact the performance of a dataframe system. The usage of dataframes has certain characteristics that differ from that of users of other scalable data processing tools like relational databases. For example, dataframe users construct queries in an incremental, iterative, and interactive fashion. This alone presents multiple challenges not present in relational databases, especially around query planning and query optimization.

In Chapter 3, we propose a candidate data model and algebra for dataframes, which lay the theoretical foundation for the rest of the paper. While dataframes have roots in both relational and linear algebra, they are neither tables nor matrices. For example, dataframes are ordered like matrices, but support more than one data type like relational databases. We will exploit the similarities and differences to enable us to define both relational and linear algebra operators in our dataframe algebra. The nuances of the challenges in supporting the unique properties of the dataframe are explored in depth, with a set of potential solutions for the unsolved dataframe challenges.

In Chapter 4, we present the architectural and design requirements of dataframe systems. This chapter does not offer an implementation, but instead proposes an architecture that other systems can implement to gain the full benefits of the dataframe algebra and data model. This chapter explores in detail what considerations may influence such an implementation. The focus in this chapter is on the nuances of the algebraic operators and implementation considerations.

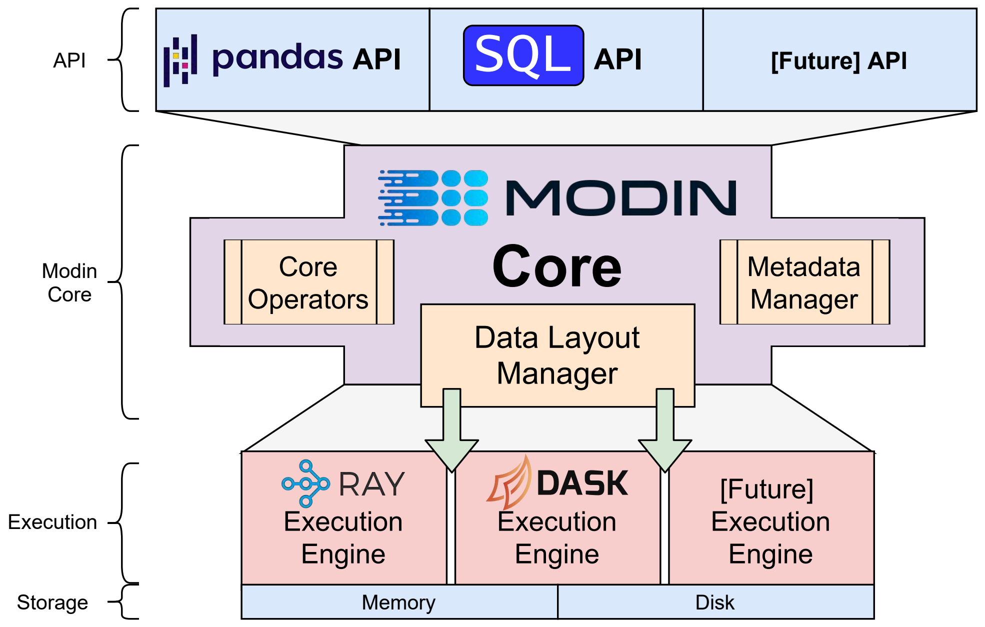


Figure 1.2: The Modin architecture

Chapter 5 presents our reference implementation Modin and the rules we use to determine the optimal parallelism in this implementation. The architecture of Modin is shown in Figure 1.2 and is designed to support multiple execution backends. This design decision stems from the goal of supporting data science in all environments, such that data scientists can use the same notebook with Modin in different operating environments with the same results. Modin exposes the familiar pandas API to users by translating pandas operators into the underlying algebraic implementation that we discuss in Chapters 3 and 4. We also discuss low level implementation decisions, with an additional focus on how we manage metadata in Modin. Each operator manipulates the metadata in specific ways, which we explain in Chapter 5. We discuss how we overcome many of the challenges in having a distributed implementation of a dataframe, including keeping certain metadata close to the user (e.g., row and column labels) while also trying to keep the labels close to the data for when they are inevitably operated on as data. Modin's impact in the open source community can be measured by the number of GitHub stars (6,000+) and installs (over one million) [73]. The impact of Modin shows that there is a significant need for distributed dataframe implementations that preserve semantics.

Chapter 6 is dedicated to evaluating different components of Modin and comparing against established data processing systems. We first do a functional analysis of Modin compared to Dask Dataframe [30] and Koalas (Spark) [63]. This functional analysis looks at how much of the pandas API is covered by each implementation. There is a subsequent discussion on the limitations of the architectures of each of the implementations evaluated here (Modin, Dask Dataframe, and Koalas) which focuses on how each implementation could change to improve the coverage of the pandas API. Following this discussion, we examine the performance of Modin on a variety of workloads and queries. We compare Modin to Dask Dataframe and Koalas, and use pandas as a baseline. Modin is up to 100x faster than Koalas, 50x faster than Dask Dataframe, and up to 50x faster than pandas on a variety of workloads and query types. With these metrics, we examine how Dask Dataframe and Spark could potentially take advantage of some of the architectural decisions in Modin to improve performance.

In Chapter 7, we do a case study on the architecture and potential optimizations for handling some of the more unique behavior of dataframe users. We propose an approach called *Opportunistic evaluation*, which leverages the user's think time to make progress on dataframe statements. Traditionally, systems have queued up operations and executed queries *lazily*, which allows for query planning and query optimization, but leaves the CPU idle for cycles that the user is spending thinking. Opportunistic evaluation leverages this think time to ensure that the CPU is working toward results to save the data scientist time. The first step in determining the potential efficacy of an approach that leverages think time is to determine how much time users actually spend thinking. We analyzed notebook traces from a corpus of Data 100 students and found that there is often significant think time spent at key places in the notebook, usually around difficult to figure out tasks. We then crafted an approach to leverage this think time that ended up being 8x faster than lazy evaluation. The results here show that traditional computation paradigms are not optimal in interactive settings, and opportunistic evaluation is a good start toward enabling efficient interactive data science.

Chapter 8 is a discussion on our experience building a successful open source community. We provide a set of recommendations based on what we did right and wrong. There is a focus on what to expect in doing so as a grad student and how it may impact other aspects of graduate school.

We begin with the requirements of a dataframe system.

Chapter 2

The Requirements of a Dataframe System

2.1 Chapter Overview

In this chapter, we discuss the requirements of a dataframe system. We will use these requirements as the basis for the architecture we present in Chapter 4 and the system we present in Chapter 5. The requirements that we outline here are based on how users interact with dataframes. We begin with a high level look at the typical data science lifecycle for most organizations in Section 2.2. In Section 2.3, we provide a motivating example that will be used in subsequent chapters to provide a simplified model of a typical dataframe workload. In interactive workflows, dataframe users often spend a large amount of time inspecting results and thinking about what they should do next. In Section 2.4, we discuss the challenges and opportunities of user think time. In Chapter 7, we are able to exploit think time to make progress on longer queries, reducing the overall time the user spends waiting on results. When inspecting results, the prefix (first k lines) and suffix (last k lines) are typically inspected to ensure that that a series of operators has executed as expected. We explore the unique challenge in dataframe prefix and suffix computation in Section 2.5. Next, in Chapter 2.6, we discuss incremental dataframe query construction, and how users compose dataframe workflows. Composability is one of the most important features of the dataframe, because it allows users to incrementally test and build larger workflows in pieces. The debugging of dataframe queries is discussed in Section 2.7. Often users will rerun the same query multiple times, which presents interesting opportunities to reuse previous results. To frame the overall problem statement, we now discuss the data science lifecycle.

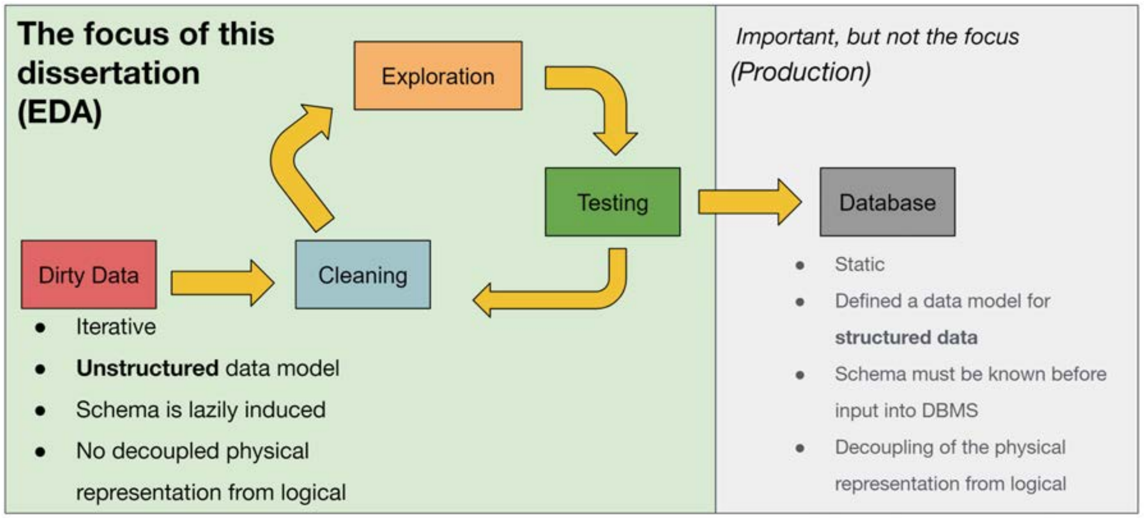

The focus of this dissertation (EDA)
Exploration
Testing
Dirty Data
Cleaning
Iterative
Unstructured data model
Schema is lazily induced
No decoupled physical representation from logical
Important, but not the focus (Production)
Database
Static
Defined a data model for structured data
Schema must be known before input into DBMS
Decoupling of the physical representation from logical

Figure 2.1: A simplified view of the data science lifecycle.

2.2 The Data Science Lifecycle

Data scientists often operate or iterate on new or unknown datasets–in various degrees of cleanliness–to extract insights or value from them. This iteration cycle is shown in Figure 2.1. Typically, as datasets are progressively cleaned and analyzed, they may be deposited in an optimized relational database for static querying, but this can only be done after the data is well-structured and the schema is defined. Given their capabilities in processing large datasets efficiently, one may be ask why not use relational databases during the earlier stages of cleaning and analysis. Unfortunately, relational databases have notable limitations when it comes to "quick-and-dirty" exploratory data analysis (EDA) [118]. 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. Due in part 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 structures and well-suited to developer and data scientist workflows, including REPL-style imperative interfaces and data science notebooks [85].

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

- an intuitive data model that embraces an implicit ordering on both columns *and* rows 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.

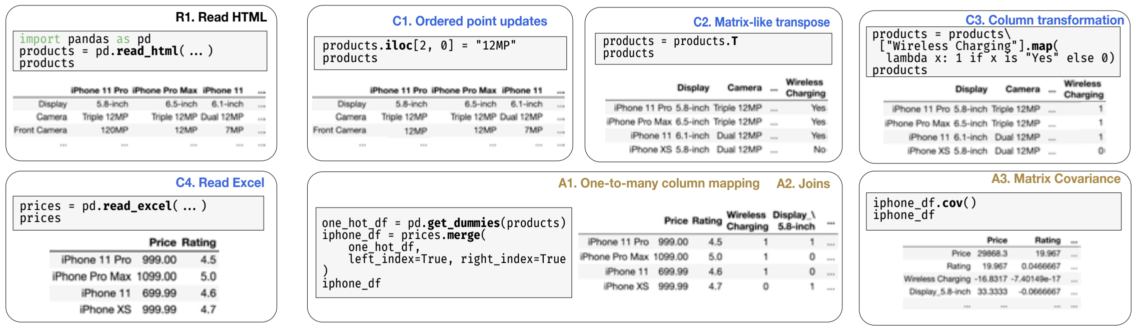
Characteristics such as these have helped dataframes become incredibly popular for EDA. The dataframe abstraction provided by pandas within Python (`pandas.pydata.org`), has, as of 2020, been downloaded over 300 million times, served as a dependency for over 222,000 repositories in GitHub, and accumulated more than 25,000 stars on GitHub. Python's own popularity has been attributed to the success of pandas for data exploration and data science [124, 89].

2.3 Motivating Example

We now discuss a motivating example that represents a data scientist's dataframe usage. In Figure 2.2, we show the steps taken in a typical workflow of a data scientist exploring the relationship between various features of different iPhone models in a Jupyter notebook [85] using pandas. The data scientist in this case has just been given access to this dataset and has no knowledge of what the data contains.

Data ingest and cleaning (Extract-Transform-Load). Initially, the data scientist reads in the iPhone comparison chart using `read_html` from an e-commerce webpage, as shown in R1 in Figure 2.2. 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 data scientist identifies a sequence of actions for cleaning their dataset:

- C1 [Ordered point updates]: The data scientist 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 data scientist transposes the dataframe (via `T`) so that the rows are now products and columns features, and then inspects the output.
- C3 [Column transformation]: The data scientist further modifies the dataframe to better accommodate downstream data processing by changing the column "Wireless Charging"


R1. Read HTML
C1. Ordered point updates
C2. Matrix-like transpose
C3. Column transformation
C4. Read Excel
A1. One-to-many column mapping
A2. Joins
A3. Matrix Covariance

```
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    ...
         ...           ...             ...        ...    ...
```

```
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    ...
         ...           ...             ...        ...    ...
```

```
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
```

```
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
```

```
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
```

```
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  Display_\   ...
                                 Charging   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   ...
```

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

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 data scientist loads price/rating information by reading it from a spreadsheet into `prices` and then examines it.

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

- A1 [One-to-many column mapping]: The data scientist 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 data scientist then verifies the output.
- A3 [Matrix Covariance]: With all the relevant numerical data in the same dataframe, the data scientist 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. 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 show in Figure 2.2, i.e., 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 are 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.

2.4 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 [31, 10] (See Section 3.6), 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 drawback of this approach is that the user must explicitly request `y` in order to realize that there is a potential bug in `x`.

Furthermore, neither the lazy nor the eager mode takes advantage of the fact that the users spend time thinking between steps, during which the system is idle. We 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 users' long think time between operations in exploratory analysis [11]. In Chapter 7 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 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 2.6).

2.5 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 2.2. 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 ordering, is important for validating the results of queries that manipulate and reshape 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 [121, 120] for `LIMIT` queries [57], or for representative tuple identification [106], but a key difference in dataframes is that order must be preserved (so "any-k" result tuples will not suffice [57]), 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 [120], 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 where 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.

Since the top and bottom k rows are often the only results inspected for dataframe queries, we may benefit from materializing additional intermediates or supporting indexes to retrieve these rows efficiently. 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 2.4. We may also be able to exploit approximate query processing to produce the prefix/suffix early for blocking operators [34, 5, 84, 47, 130]. 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 [58, 83, 66, 7]; 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 [93].

While it is well known that some aggregates like `MAX` cannot be approximated [75], even blocking operators such as `SORT` can return early approximate results, using results from the 1990s [94]. 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 [92], web-database hybrid [39], and crowdsourcing [82] contexts, but not, as far as we can tell, for a group-by aggregation setting. This idea could also be applied to operators 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. [65], 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 [56] 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.

2.6 Incremental Query Construction and Composability

In addition to challenges around enabling immediate feedback, query optimization is further complicated by the need to frequently evaluate and display results for intermediate sub-expressions (i.e., the results of statements) over the course of a session (see also Section 2.4). 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 sub-expression. However, since user statements often build on others, we can jointly optimize across these statements and resulting sub-expressions, sharing the 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.

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 2.5), many statements are often 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., [104, 46, 38, 98]. 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" [38], and "the synchronization of the execution of queries with common subexpressions when queries are submitted at different moments in time" [38]. 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 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. Therefore, we simply need to construct a query plan wherein sub-plans that correspond to intermediate dataframe results are materialized as by-products. These intermediates are also likely to be reused by the user in the future. This presents an interesting conundrum because ensuring that the sub-plan results are materialized "along the way" may result in suboptimal overall plan selection (e.g. if the user cares more about the final dataframe than intermediates). 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 explicitly requested. 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 correspond 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.

2.7 Debugging & Building Queries

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 [128]. 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 [100]. 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 fan-outs; 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 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 [119], which is a query engine for static 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.

In the next chapter, we describe a theoretical basis on which we can build a dataframe system. We aim to preserve the semantics of dataframes and define a dataframe data model and algebra that we can use to build a dataframe system that scales.

Chapter 3

Dataframe Theoretical Foundation

3.1 Introduction

As we described in Section 2, dataframes are not well understood in scientific literature. In this chapter, we explore the history of dataframes, and propose a data model and algebra as a solid theoretical basis that practical systems can implement. We note that this is the first candidate data model and algebra for dataframes presented in the literature to date.

3.2 History of the Dataframe

The history of dataframes begins with the S programming language. 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 [19]. 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 [20], 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.

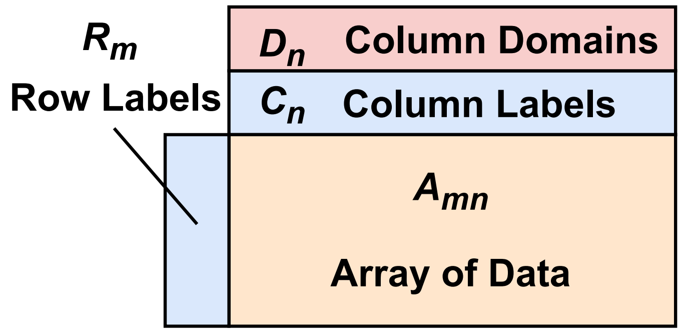


Figure 3.1: The Dataframe Data Model

3.3 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. [3, 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** $S : (\Sigma^*)^m \rightarrow Dom$ 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 3.3.1. *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 3.1. In our example of Figure 2.2, 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 2.2, 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.

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** [3]). 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 3.4.

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 schemata (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, discussed in Section 3.4. 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. One way our data model treats rows and columns differently is via D_n, which is associated only with columns rather than rows. While we could have equivalently defined a row-level schema S_m, in many cases, e.g., when each of the columns have distinct types, this will be overkill as the typing for each row will correspond to an array of different types (or more compactly represented as Σ^*). That said, specific dataframe systems may certainly keep track of row schemata, especially when the types of all values in a row are the same, since it will be useful when combined with transpose.

Despite the notational symmetry between rows and columns, D_n introduces a *schematic asymmetry* that we will need to reason about. Consider a schema like D_n = [**category**,

Table 3.1: Dataframe Algebra Description.

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

int, **float**]. Note that the column types differ, but the type of each row is the same (namely [**category**, **int**, **float**]). Moreover, each column has a single domain (an atomic type), but each row has a vector of domains (a tuple type). Once seen through the lens of a schema and its parsing functions, rows and columns are quite different.

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 2.2 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.

3.4 Dataframe Algebra

While studying pandas, 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 3.4 that it can be used to express `pivot`. 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. [51, 52] 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.

We list the algebra operators we have defined in Table 4.1: 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 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.

We succinctly describe the new operators as well as highlight any deviating semantics of `GROUPBY` and `WINDOW`. The output schema for most other relational operators can be carried over from the inputs (indicated as *static* in Table 4.1).

It's important to note that languages choose different approaches to inferring the schema after a `TRANSPOSE` which can have 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

Table 3.2: Dataframe Algebra Data Manipulation and Order. $\dagger$: Ordered by left argument first, then right to break ties. $\diamond$: Order of columns is inherited from order of rows and vice-versa.

Operator	**(Meta)data**		**Schema**	**Origin**	**Order**
SELECTION		$\times$	static	REL	Parent
PROJECTION		$\times$	static	REL	Parent
UNION		$\times$	static	REL	Parent†
DIFFERENCE		$\times$	static	REL	Parent†
CROSS PRODUCT / JOIN		$\times$	static	REL	Parent†
DROP DUPLICATES		$\times$	static	REL	Parent
GROUPBY		$\times$	static	REL	New
SORT		$\times$	static	REL	New
RENAME	$(\times)$		static	REL	Parent
WINDOW		$\times$	static	SQL	Parent
TRANSPOSE	$(\times)$	$\times$	dynamic	DF	Parent$^\diamond$
MAP	$(\times)$	$\times$	dynamic	DF	Parent
TOLABELS	$(\times)$	$\times$	dynamic	DF	Parent
FROMLABELS	$(\times)$	$\times$	dynamic	DF	Parent

of "crosstabs" data. In step C2 in our example in Figure 2.2, 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, 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").

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 columns or 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 data layout manipulation costs.

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 2.2). 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 $f : D_n \rightarrow D'_{n'}$, 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.

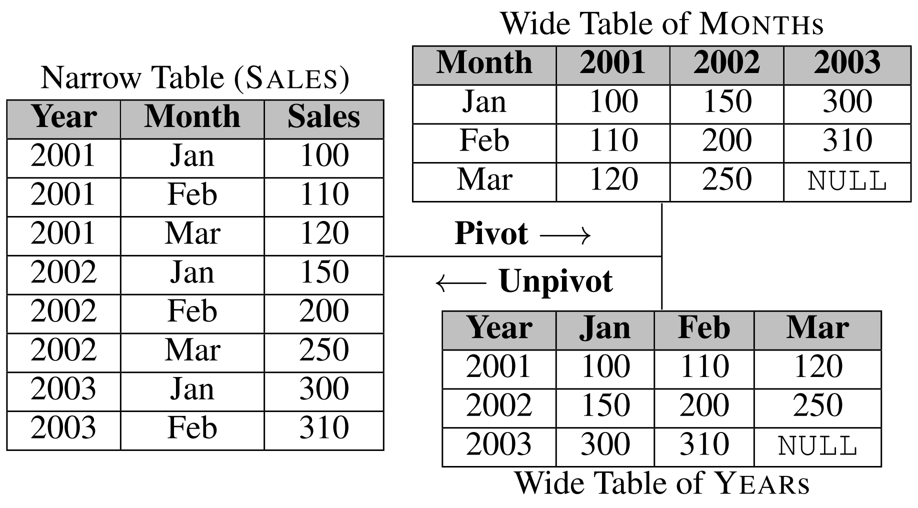
FromLabels. `FROMLABELS` creates a new dataframe with the row labels inserted into the array A_{mn} as a new column of data at position 0 with a provided column label. The data type of the new column starts as $null$ until it can be induced by the schema induction function S. The row labels of the resulting 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. An optimized query planner will need to avoid materializing the sub-dataframes when possible.

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.



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

Wide Table of MONTHs

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

Pivot ⟶
⟵ **Unpivot**

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

Wide Table of YEARs

Figure 3.2: Pivot table example, reproduced from [26], demonstrating pivoting over two separate columns, “Month” and “Year”.

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.

Algebra Examples

Pivot Case Study

To demonstrate the expressiveness of the algebra above, we show how it can express `pivot`, which is particularly challenging in relational databases due to the need for relations to be declared schema-first [126, 26]. 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.

The `pivot` operator (Figure 3.2) elevates a column of data into the column labels and creates a new dataframe reshaped around these new labels. The `pivot` operator has been described and implemented in relational systems [126, 26] but it is simpler to express in the algebra from Section 3.4.

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 3.3.

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.

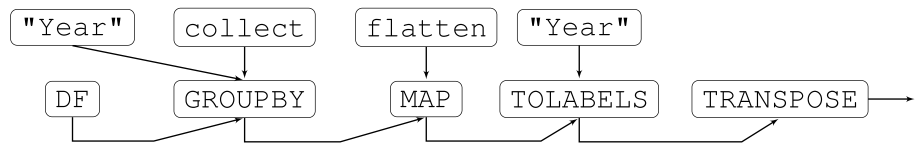


Figure 3.3: Logical plan for pivoting a dataframe around the "Year" column using the dataframe algebra from this section.

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

3.5 Data Model Challenges

Supporting the dataframe data model and algebra from Section 3.3 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.

Flexible Schemas, Dynamic Typing

Major challenges arise from the flexible nature of dataframe schemas. Dataframes require more than data; as noted in Section 3.3 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 column must be known before we can parse the value of any cell in that column. Mitigating the costs inherent in flexible schemas and dynamic types therefore presents a major challenge for dataframes.

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.

Rewrite Rules for Schema Induction

Due to their flexible schemas, dataframes support the 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.

In some cases, schema inference rules might be able to avoid the application of S altogether. For 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 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 3.5) 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.

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 (Chapter 2).

In some cases, it is reasonable to expect that a data scientist 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*.

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, 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 measures 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 if 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 into the `MAP`.

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, is crucial for the development of a full-fledged dataframe query optimizer.

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 3.4, 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.

Order is Central

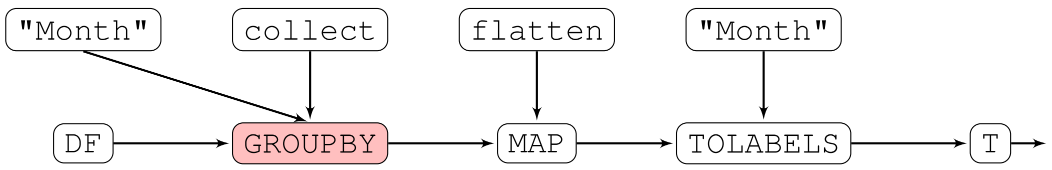
The order of a dataframe is determined by the order of ingested data. For example, a CSV file ingested as a dataframe would have the same row and column order as the file. This ordering is crucial for the trial-and-error-based interaction between a user and a dataframe system. Users expect to see the 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` expect a specific order for the rows (`WINDOW`) and columns (`MAP`). Perhaps most challenging is the frequency at which the order can be changed, as each operator has a deterministic output order (shown in Table 4.1). 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.

Current dataframe systems such as pandas physically store the dataframe in the order defined by the user and do not implement 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.

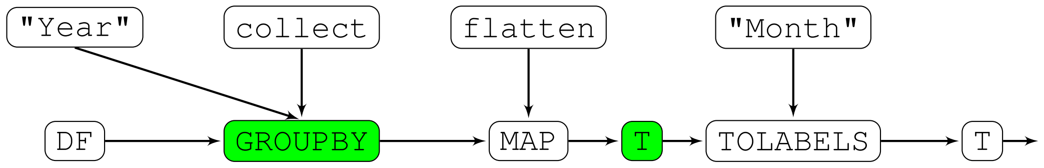
To ensure correct semantics while respecting physical data independence, we must devise a means to capture ordering information, either tracked as a separate "order column" if it is not implied via existing columns, or recording as metadata that the dataframe must be ordered based on one or more of the 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, as discussed in Chapter 2.

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 [1], as well as those from scientific computing [23], array databases [101], or spreadsheets [13]. 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, column) 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.

The order of the dataframe also creates some interesting new challenges in query planning. Operators that are commutative in the relational data model are not necessarily commutative in dataframes, e.g., `SELECTION` based on a positional predicate. Due to this, a dataframe query optimizer must now be aware of and manipulate an internal representation of the order. The added complexity of maintaining order in the query plan



(a) Original plan



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

Figure 3.4: Alternative query plans for pivoting a dataframe around the "Month" column. TRANSPOSE is abbreviated as T.

due to the presence of non-commutative operators in dataframes will be a significant challenge for dataframe systems wishing to preserve these semantics.

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 [14], 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 [1]; 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.

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) [91, 50]. 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 used to perform optimizations. To ensure these optimizations are harmonious with respect to `TRANSPOSE`, we can do a *logical* `TRANSPOSE` “pull-up”. The proposed rewrite delays transpose in the physical plan as much as possible, since it will often destroy many data layout optimizations that would otherwise apply.

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 3.4, 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 `Month`s as columns, we can either use the original plan (Figure 3.4a) 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 3.4b). 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" [103] or "hash teams" [43], expanding the rules for query optimization.

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 [8], SchemaSQL [64] and XQuery [18]. There is significant prior work on optimizing second-order operations like the unnesting of nested data (e.g. [35, 105, 123]). A distinguishing aspect of our setting is that a dataframe operation like `TOLABELS` commonly generates a volume of schema-level metadata that is dependent on the size of the data; this raises new challenges. The closest prior work to our needs studies spreadsheet-style pivot/unpivot in databases (e.g. [26, 126]); 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 [62] or range min-max trees [76]. 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* [26], 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.

In terms of query optimization, we now have a two-dimensional estimation problem: both cardinality estimation (# of rows) and *arity estimation* (# of 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`. While techniques like hyperloglog sketches [36] could assist here, 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 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.

3.6 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 3.3 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.

Dataframe Implementations: R. The R language (and the S language before it), both support dataframes in a manner similar to pandas and can be credited for initially popu-

Table 3.3: 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 by the row labels after `TOLABELS`.

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

larizing the use of dataframes for data analysis [54]. 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 3.3, 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 [117, 125].

Dataframe-like Implementations. Some libraries provide a functional or object-oriented programming layer on top of relational algebra. These libraries include SparkSQL dataframes [88], SQL generator libraries like QueryDSL [95] and JOOQ [68], and object relational-mapping systems (ORMs) such as Ruby on Rails [99] and SQLAlchemy [12]. All of these systems share some of the benefits with respect to incremental query construction mentioned in Chapter 2. 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 3.3, 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 [63], 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.

3.7 Discussion

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. In this chapter, 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.

Now that we have a theoretical foundation for dataframes, the logical next step is to design an architecture that can support this data model and algebra. In the next chapter, we explore a general architecture for dataframes.

Chapter 4

A General Dataframe Architecture and Design

4.1 Introduction

In this chapter, we discuss a general architecture design for dataframe systems. Specific implementation details are reserved for a later chapter. In this chapter we ignore implementation details related to exposing the pandas API, and instead focus on the architecture design of a system that is able to implement the *functionalities* supported by the pandas API. While this is not the only design possible, critical components described here (e.g., dataframe algebra design) must follow the rules outlined here to be a complete implementation of a dataframe. Some components (e.g., physical layout management) are open to interpretation and modification.

To keep the chapter self-contained, we re-introduce or paraphrase some definitions here.

4.2 Dataframe data model

In this section, we introduce a set of rules and properties that dataframes must follow for the language and type system to be applicable. Modifying or removing any of these properties will disqualify a system from being designated as a dataframe. We aim to define these system properties with respect to the dataframe data model and algebra presented in Chapter 3. We note that the algebra in Chapter 3 requires some minor changes to adapt to real systems, which we discuss below. This section introduces a concrete data model for systems to implement.

A dataframe consists of four main components, each of which will be described in detail below. The four components of a dataframe are as follows:

- A collection of data, logically arranged into ordered columns and ordered rows
- A set of row labels - one per row
- A set of column labels - one per column
- A set of column types - one per column

Data, logically arranged into columns and rows. There must be some logical representation of columns and rows in a dataframe, as in a 2-dimensional array. No additional dimensions are supported in the dataframe data model.
Row labels. Row labels are a "special" column or columns of values that represent each row. Row labels must be able to contain any data type that the dataframe can support, e.g., floating point data. There is no special type designation for row labels because they can be treated as data with some operators.
Column labels. Column labels are a set of labels that each represent a single column. It is possible to have hierarchical column labels in this data model. As with row labels, there is no special type designation for column labels, and they must be able to contain any data type that the dataframe can support.
Column types (domains). Column types are a mapping of the type to each column. Any of these types may be left unspecified and inferred/coerced at runtime. In the case that implementations do not support unspecified types, the set of operations will be limited to those operations that have known output types.

Each data value can have independent data types, and they may be tracked separately from the column labels. In the case that the values have individual data types, it must fit into the type hierarchy, described below. Each data value must have a type associated with it at all times, either via some individual designation or via the column types object in the dataframe data model. Column types are queryable, so they must be structured in a way that enables the data to be filtered by the type.

4.3 Dataframe Algebra Layer Design

As we previously mentioned, the algebra presented in Chapter 3 needs some changes to adapt to real systems. In this section, we provide a concrete design for the algebra. We now describe each algebraic operator in detail, its parameters, and the implementation assumptions made. Table 4.1 shows the mapping between a high level classification of pandas behavior and the dataframe algebra syntax, with an expanded list of public pandas APIs that follow that pattern included for completeness.

Table 4.1: Dataframe Algebra mapping to pandas APIs.

Operator Pattern	**Modin Syntax**	**Exhaustive list of pandas public API**	Count
Applying a user-defined function uniformly element-wise, column-wise, or row-wise	`map(df, axis, f)`	`abs`, `astype`, `clip`, `eval`, `isna`, `isnull`, `notna`, `notnull`, `fillna`, `isin`, `apply`, `applymap`, `transform`, `fillna`, `rank`, `round`, unary version of {`add`, `subtract`, etc.}, string manipulations (`str`), datetime manipulation (`dt`), `replace`	42
Binary function between two dataframes	`map(join(df1, df2), f)`	`add`, `subtract`, `multiply`, `divide`, `eq`, `floordiv`, `where`, `update`, `corr`, `corrwith`, `combine`, `combine_first`, `cov`, `dot`, `ge`, `gt`, `lt`, `pow`, `radd`, `rdiv`, `rfloordiv`, `rmod`, `rmul`, `rpow`, `rsub`, `rtruediv`, `sub`, `truediv`	28
Indexing: Queries on the row labels	`mask(df, row_indices=query)`	`as_freq`, `asof`, `at`, `at_time`, `between_time`, `drop_level`, `drop`, `first`, `first_valid_index`, `iloc`, `last`, `last_valid_index`, `loc`, `mask`, `pop`, `get`, `head`, `tail`, `iat`, `lookup`, `resample`, `sample`, `select`, `take`, `truncate`	26
Treating metadata as data, metadata manipulation, and querying	`to_labels(map( from_labels(df), f), label)`	`add_prefix`, `add_suffix`, `swap_level`, `melt`, `reorder_levels`, `shift`, `slice_shift`, `tshift`, `tz_convert`, `tz_localize`	10
Reshaping, transposing, pivot	`[map/groupby]( transpose(df), f)`	`pivot`, `pivot_table`, `stack`, `unstack`, `transpose`, `T`	6
One-hot (dummy) encoding	`transpose( to_labels( transpose(map(df, f, axis=1))))`	`get_dummies`	1
User-defined aggregation of values per-column	`reduction(df, f)`	`all`, `any`, `count`, `agg`, `aggregate`, `compound`, `describe`, `duplicated`, `equals`, `idxmax`, `idxmin`, `kurtosis`, `mad`, `max`, `mean`, `median`, `min`, `mode`, `memory_usage`, `prod/product`, `nunique`, `quantile`, `sem`, `skew`, `std`, `sum`, `var`	28
Aligning and joining two dataframes on row or column labels	`join( from_labels(df1), from_labels(df2), on="index", axis)`	`align`, `concat`, `join`, `reindex`, `reindex_axis`, `reindex_like`	6
Window user-defined functions (window size < length of dataframe)	`window(df, axis, f, size)`	`bfill`, `cumsum`, `cumprod`, `cummax`, `cummin`, `diff`, `ewm`, `expanding`, `ffill`, `fillna`, `interpolate`, `nLargest`, `nsmallest`, `pct_change`, `rolling`	15
Conditional filter	`filter(df, f)`	`dropna`, `drop_duplicates`, `filter`, `query`	4
Queries on the schema of the dataframe	`filter_by_types(df, types)`	`select_dtypes`	1
Type Inference and induction	`infer_types(df, columns)`	`infer_objects`, `convert_objects`	2
Column/row insertion and assignment, appending columns/rows	`concat(df1, [df2])`	`append`, `assign`, `concat`, `insert`	4
Groupby with a user-defined aggregation or function	`groupby(df, by, f)`	`groupby`	
Join on an attribute	`join(df1, df2, condition, how)`	`merge`, `merge_asof`	2
Sorting on labels or column values	`sort_by(df)`	`sort_index`, `sort_values`	2
Expand the number of rows or columns	`explode(df, f)`	`explode`	1

Differences with Dataframe Algebra

Dataframe algebra, as described in Chapter 3, is a minimal set of operators that represent everything possible in popular dataframe systems like pandas and R. The algebra, however, relies heavily on `TRANSPOSE` to perform operations along either axis. While this does make the algebra compact and expressive, repeatedly transposing the data in practice is extremely inefficient [87]. Instead, a dataframe implementation should allow all operators to be applied over either axis with an `axis` argument to avoid a transposition.

Notice that the API presented here differs slightly from the algebra presented in Chapter 3 in the interest of being explicit and avoiding having to inspect into opaque user-defined functions. Consider the case of the algebraic operator `MAP`: since a valid `MAP` can return one or more rows for each input row, it would be beneficial for the query optimizer to know upfront to expect the number of output rows to match the input row count. Rather than make this a parameter to the `MAP` implementation, we chose to separate these behaviors into two different operators in the implementation to encourage developers to choose correctly.

`mask`

The `mask` operator is equivalent to a combination of `SELECT` and `PROJECT` from SQL. Dataframes allow users to perform selection and projection on the row and column number (positional notation), in addition to the row and column labels (named notation), so this must be exposed at this layer. While it would be possible to convert both positional and named notation to the same representation at a higher layer and only expose a single mask notation at this layer, we felt it was critical to the performance of the implementation to keep them separate. Named notation does not inherently rely on any order, but positional notation does. If the data is not ordered before a named notation mask operator, it can remain unordered for the execution. If the data is not ordered before a positional notation mask operator, it must be ordered first so that the correct rows or columns are selected.

The `mask` operator in the dataframe algebra allows the fusion of traditionally separate `SELECT` and `PROJECT` operators into a single operator. This simplifies query execution and planning, especially with the presence of a `TRANSPOSE` operator.

The interface of `mask` is as follows:

```
def mask(
    df,                    # input dataframe
    row_indices=None,      # one or more row labels
    row_positions=None,    # one or more row positions
    col_indices=None,      # one or more column labels
    col_positions=None,    # one or more column positions
) -> Dataframe
```

The `row_indices` and `col_indices` parameters are for providing named notation, where `row_positions` and `col_positions` are for providing positional notation.

This design does not allow users to pass both positional and named notation for the same axis (columns or rows), even though this is not explicitly forbidden by the algebra. The design does, however, allow the mixing of positional and named notation across axes in a single operator, which is useful for operator fusion during query planning.

filter_by_types

One of the primary differences between dataframes and relational tables is the ability to query metadata. For example, suppose a user wanted to compute the median for each column that contained `DATETIME` data. The `filter_by_types` operator enables the user to specify a type or set of types by which to filter the columns. Internally, this filter may be encoded as a `mask` for the purposes of operator fusion and query planning.

The interface of `filter_by_types` is as follows:

```
def filter_by_types(df, types) -> Dataframe
```

The `types` parameter is an iterator of data types, usually provided as a string name for the type (e.g., `int` or `float64`). A dataframe implementation should be able to query against the internal representation of data types with these provided types.

map

The `map` operator applies a user-defined function row-wise (or column-wise if `axis=1`). The user-defined function should not change the number of rows or columns, and does not change the row labels or column labels. There are no restrictions on the output data types or how the data may be transformed. The output dataframe's column labels must remain the same as the input dataframe's, with the exception of the added columns. The row labels may not be changed from the input dataframe's.

There are multiple examples of `map` style operators in the pandas API. `DataFrame.isna` is an example of a `map` that does not change the output number of columns, but will change the data types if they were not already boolean.

The interface of `map` is as follows:

```
def map(
    df,         # input dataframe
    axis,       # 0 (rows), 1 (columns), or None (any)
    function,   # callable function
    result_schema=None
) -> Dataframe
```

The `axis` argument is for determining which axis to perform the `map` operation over, and can be 0, 1, or `None`. When `axis` is 0, the function is applied across each row, or column if `axis` is 1. When `axis` is `None`, the `map` operation can be applied across either axis. The `axis=None` behavior is unique to `map`. The `function` argument is a callable function that is applied. The optional `result_schema` argument is an iterator of data types that represent the types of the output dataframe.

filter

The `filter` operator is equivalent to a `SELECT` query which does not solely rely on the dataframes's labels or position. It is also important to note that this operator also supports the `axis` argument, so columns may be dropped conditionally based on the data they contain. The function argument may only access values within the same row (column if `axis=1`), but otherwise has no restrictions outside of the boolean return type.

Filtering rows or columns by the values they contain is a common task in data preparation and cleaning.

```
def filter(df, axis, condition) -> Dataframe
```

One example of the power of this operator is in a case where the user wants to anonymously filter across columns that do not meet a threshold of non-null values.

The `axis` argument is can be 0 or 1. When `axis` is 0, the condition is applied across each row, or column if `axis` is 1. The `condition` argument is a filter function: `row -> boolean` where rows that are evaluated to `TRUE` are dropped and rows that evaluate to `FALSE` are kept.

explode

`explode` is similar to a `map`, but it can increase the number of rows instead of columns (or vice-versa if `axis=1`). The concept of `explode` was not present in the algebra presented in Chapter 3 because it was explicitly introduced into the pandas API after the publication date. Despite the fact that it is relatively new in pandas, many SQL systems have a concept of "exploding" columns [97], which converts a single column containing an array or some other collection into an individual row for each value in the collection, replicating the values in the other columns in the row.

Notice that `explode` is still defined in terms of rows in this algebra implementation, which is the finest-grained level of parallelism possible. This implementation does not require that there be a column of arrays, nor does it place a hard limit on the number of rows that can be generated from each row. This is an extension on the traditional view of SQL `EXPLODE` implementations, and gives the developer more freedom. The `explode` operator does not allow the removal of rows - dropping rows or columns is only allowed via `filter` and `mask`. The `explode` operator also may not change the number of columns (rows if `axis=1`), only one axis may be expanded at a time. The user-defined function for `explode` does not have any restrictions other than that; the types may be transformed from the input dataframe.

It is common in real-world datasets to have a column containing arrays of values, and exploding columns into discrete values is a common data preparation task.

The signature for `explode` is as follows:

```
def explode(
    df,          # input dataframe
    axis,        # 0 (rows) or 1 (columns)
    function,    # callable function
    result_schema=None
) -> Dataframe
```

The `function` argument is a callable function that accepts a row and outputs multiple rows (columns if `axis=1`). The optional `result_schema` argument is an iterator of data types that represent the types of the output dataframe.

`window`

The `window` operator performs a sliding-window operation, but output dataframe's shape must match the input dataframe's shape (number of rows and columns) for this specific operator. In the case of a `window` with a `GROUP BY` style aggregation, we have a separate operator, `window_reduce`, explained below. These operators are separated due to the opaque nature of the function being passed as discussed above.

The sliding window function that is provided to this operator may only access values in the same column (row with argument `axis=1`). Drawing from the pandas API, consider the case where the end-user wants to replace `NaN` values via a back-filling function, up to a certain limit. In that case, the developer would use a `window_size` equal to the limit.

```
def window(
    df,             # input dataframe
    axis,           # 0 (rows) or 1 (columns)
    function,       # callable function
    window_size,    # number of input rows for `function`
    result_schema=None
) -> Dataframe
```

The `function` argument accepts a subset of the column values equal to the `window_size` argument, and outputs an identical number of values. The `window_size` argument is an integer that represents the number of values that the `function` argument accepts. The optional `result_schema` argument is an iterator of data types that represents the types of the output dataframe.

window_reduction

The `window_reduction` operator executes a sliding window operator that acts as a `GROUPBY` on each window, which reduces down to a single row (column) per window. It can be performed along either the column or row axis, specified by the `axis` argument. This operator is deliberately specified separately from `window` to optimize the metadata during query planning and avoid passing around state or inspecting the reduction function.

Like `window`, the sliding window of `window_reduction` moves in the order of the data. The user-defined reduction function: `column -> scalar` must reduce each window's column (row if `axis=1`) down to a single value. This differs from the `groupby` behavior, described below.

```
def window_reduce(
    df,                    # input dataframe
    axis,                  # 0 (rows) or 1 (columns)
    reduction_function,    # callable function
    window_size,           # number of input rows
    result_schema=None,
) -> Dataframe
```

The `reduction_function` argument accepts a column of values and outputs a column of values . The `window_size` argument is an integer that represents the number of values that the `function` argument accepts. The optional `result_schema` argument is an iterator of data types that represent the types of the output dataframe.

groupby

The `groupby` operator will generate groups based on the values in an input column or columns. Typically, an aggregation is performed during the groupby, but other operations are also possible, e.g. `fillna`, where `NaN` values may be filled based on some properties of the group.

`groupby` accepts a user-defined function: `dataframe -> dataframe` that operates on each group independently. Like SQL, no communication between groups is allowed in this algebra design. The number of rows (columns if `axis=1`) returned by the user-defined function passed to the `groupby` may be at most the number of rows in the group, and may be as small as a single row.

Unlike the pandas API, an intermediate "GROUP BY" object is not present in this algebra. One argument for exposing such an object is for reusing the `groupby` parameters to run different aggregation functions. However, while working with users, we found that reusing the `DataFrameGroupBy` object is not common, and presenting such an object in the algebra does not appear to provide any significant value, but instead exposes more opportunities for bugs. During the design of this architecture, we discovered cases in pandas where the state of the `DataFrameGroupBy` object was mutated during execution, which would cause different output results depending on the order by which aggregation functions were applied.

The operator signature for `groupby` is as follows:

```
def groupby(
    df,          # input dataframe
    axis,        # 0 (rows) or 1 (columns)
    by,          # column(s) to group by
    operator,    # callable algebraic operator
    result_schema=None
) -> Dataframe
```

The `by` parameter is one or more column labels to use for the group by. The `operator` parameter is the operation to carry out on each of the groups. The `operator` is another algebraic operator with its own user-defined function parameter, depending on the output desired by the user.

For example, a user may wish to replace null values within each group with a back-fill function, such that the null value replacements only come from values within that group. This would use the `map` operator from Section 4.3.

`reduction`

The `reduction` operator is a per-column aggregation, where each column reduces down to a single value.

In the pandas API, this behavior is directly reflected with multiple aggregations, including `pandas.DataFrame.count` and `sum`. Unlike SQL systems, each column does not need to be specified for the aggregation function, it is automatically applied to each column. The only restriction on the user-defined function for `reduction` is that it must reduce to a single value. For some tasks, e.g. counting the number of non-null values in each column, it is possible to arrive at the result via an efficient tree reduction computation pattern. However, some user-defined functions may require access to the entire column to be computed correctly, e.g. computing the median. Every reduction that is possible via tree reduce is also possible with access to the entire column, so we expose a flag that allows the developer to enable the tree reduce paradigm to be applied for efficiency. We chose a flag instead of a completely new operator because the result will always be correct if the user-defined function is applied with access to the whole column, and minimizing the number of operators exposed by the algebra is central to our design principles.

```
def reduction(
    df,          # input dataframe
    axis,        # 0 (rows) or 1 (columns)
    function,    # callable function
    tree_reduce=False,
    result_schema=None,
) -> Dataframe
```

The `function` parameter is a callable function that takes a number of rows and outputs a single row. For example, the `function` argument can sum and output the values of all rows in each column. The `tree_reduce` flag, as mentioned above, signals that the `function` can be applied via the `tree_reduce` algorithm.

infer_types

This operator was not explicitly exposed by the algebra defined in the previous paper, but we have found it to be a useful operator for implementations that throw a runtime type error rather than infer the types. Additionally, this gives the developers the ability to force the materialization of the unspecified types for use with the user-defined function definitions supported in many of the algebra operators. This operator serves two main functions: (1) to look at every type and find the common type among them all and (2) to coerce all of the types, if needed, to the same overall type or throw a type error in the case of incompatible types. The coercion rules are left to the implementation, and we omit them from this paper in the interest of space.

In the pandas API, there is an operator called `infer_objects` that behaves very similarly to the `infer_types` operator from this algebra. In pandas, the default type is called `object`, which is the parent data type of all types. If we interpret `object` as an unspecified type, then one key difference in the behaviors between the implementation of `infer_types` and `infer_objects` from pandas is that in this design, the operator can be applied to any subset of columns in the interest of performance and pandas will attempt to infer the types of every column with “unspecified” types. We expect this operator to be extremely expensive in terms of execution runtime and communication, so any implementation should try to put off inferring types for as long as possible. Another key difference between the algebra design and pandas is that this operator does not modify the input dataframe’s schema in-place. The data and metadata of a dataframe is immutable, so this operator will return a new dataframe with the inferred schema applied to the columns selected.

```
def infer_types(df, column_list) -> Dataframe
```

The `column_list` parameter is a list of column labels representing the columns on which to infer and induce the types.

join

The `join` operator is analogous to `JOIN` in SQL. This is the algebra entry point for the following join types: `INNER JOIN`, `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, and `FULL OUTER JOIN`. Other join types are not explicitly supported by the current design, though this design may be extended to support additional join types if an implementation supports it, e.g. `NATURAL JOIN`.

The signature of the `join` operator is as follows:

```
def join(
    df,          # input dataframe
    axis,        # 0 (rows) or 1 (columns)
    condition,   # callable condition function
    other,       # the other dataframe
    join_type,   # the type of join, e.g. "inner"
) -> Dataframe
```

The `condition` parameter is a function that is applied to determine which rows should be joined. The `condition` can be a simple equality, e.g. `"left.col1 == right.col1"` or can be arbitrarily complex. The `other` parameter is the data to join with, and is considered the “right” dataframe where `df` is considered “left”. The `join_type` is the type of join to perform.

concat

The `concat` operator is analogous to a `UNION` in SQL, appending the rows of identical column labels from two dataframes, however it has some key differences. First, `concat` does not require that both dataframes have the same number of columns (rows if `axis=1`), and there are no restrictions between the labels of the two dataframes, even their types may be different. Another key difference is that the `concat` operator can accept multiple dataframes at a time.

From pandas, this is represented by the `pandas.concat` utility. In practice, it is common to have multiple different data sources that need to be concatenated for a larger analysis, as opposed to `join` which typically only involves two tables. There are fixed overheads to the `concat` operator, so this algebra places no limit on the number of dataframes that may be concatenated in this way.

The signature of `concat` is as follows:

```
def concat(
    df,        # input dataframe
    axis,      # 0 (rows) or 1 (columns)
    others,    # one or more dataframes
) -> Dataframe
```

The `others` parameter is one or more dataframes to concatenate with `df`.

`transpose`

The `transpose` operator logically swaps the row and column axes, but not necessarily physically. Since it is such an expensive operation, we try to delay it until absolutely necessary, which is part of the motivation for the `axis` argument on each operator within the algebra definition.

Pandas has a `transpose` operator for its `DataFrame` object which is commonly used in cases where data has been laid out incorrectly from the source.

The signature of `transpose` is as follows:

```
def transpose(df) -> Dataframe
```

`to_labels`

`to_labels` transforms the input dataframe to replace the row labels with one or more columns of data. In the case that multiple columns are selected as new labels, a hierarchical set of labels is created in the order specified in the operator's parameters. Relational databases do not have an equivalent abstraction to the row labels in the dataframe data model, so it cannot be expressed purely in SQL.

The signature for `to_labels` is as follows:

```
def to_labels(column_labels) -> Dataframe
```

The `column_labels` parameter is one or more column labels that become the new dataframe's column labels.

`from_labels`

The `from_labels` operator moves labels into the data at position 0 and sets the row labels to the positional notation. In the case that the dataframe has hierarchical labels, all label “levels” are inserted into the dataframe in the order they occur in the labels, with the outermost level or labels inserted into position 0 and subsequent levels inserted at the following positions.

The signature for `from_labels` is as follows:

```
def from_labels(df) -> Dataframe
```

sort_by

`sort_by` is an operator that logically reorders the dataframe’s rows (columns if `axis=1` by the lexicographical order of the data in a column or set of columns. This is analogous to the `ORDER BY` SQL operator. Like its SQL companion, in the case that multiple columns are provided, columns after the first (in the order provided) will break ties by the lexicographical order of the earlier columns. The sort may be in ascending or descending order, depending on the arguments.

In pandas, there are two ways to sort: `sort_values` and `sort_index`. The `sort_values` call has the same behavior as this algebraic operator. The `sort_index` operator sorts the by the lexicographical order of the row (column is `axis=1`) labels. The `sort_by` algebraic operator does not interact with the row or column labels of the data, so a `from_labels` operator must be invoked before the `sort_by` operator to move the labels into the data so that it may be sorted by the values in those labels. After the sort, `to_labels` can then move the columns back into the labels, which will result in matching behavior to pandas.

The signature for `sort_by` is as follows:

```
def sort_by(
    df,        # input dataframe
    axis,      # 0 (rows) or 1 (columns)
    columns,   # one or more column labels
    ascending=True
) -> Dataframe
```

The `columns` parameter is one or more column labels to sort by. The `ascending` flag specifies whether or not to sort in lexicographically ascending or descending order.

Generalization of the algebra

The design here is a generalization of what pandas can implement, and can express operations beyond the pandas API. It is intended to be able to support any dataframe API, from any system past or future. We do not intend for this design to be user-facing due to the heavy reliance on user-defined functions. Instead, we designed this architecture such that user-facing applications may be built to integrate with it.

Next, we will discuss the design of the metadata management in dataframes.

4.4 Metadata Management

The core function of the metadata management component of this design is to manage the dataframe's schema, the logical and physical order mappings, and the dataframe labels. We now discuss each of these components individually.

Schema Management

The schema in dataframes has unique properties with respect to relational databases which require extra engineering attention. In dataframes, any or all of the column's types may be left unspecified and inferred on-demand. The inference of types is extremely expensive, requiring up to two passes on the data, as described in Section 4.3. The type inference is fine-grained and can be applied at a column level, but it may be more efficient to infer the types of multiple columns that have been unspecified when the inference of one column is forced. We now go through each operator and its schema mangement requirements.

`mask` does not force the materialization of the unspecified types in the schema and the resulting types remain unchanged from the input dataframe.

`map` will not force the materialization of the types, and the user-defined function may make assumptions about the type of each column, but these assumptions are not guaranteed by the algebra. The schema's data types may be manually inferred and queried by the user-defined function passed to `map`: schema inference is invoked through the `infer_schema` operator in this algebra and the schema may be directly accessed.

The `filter` operator does not require materialization of the data types in the schema. Like `map`, the condition function may make assumptions about the data types that are unspecified and the developer may force the inference of types via the `infer_types` operator. The types of the resulting dataframe remain the same as the input dataframe.

The `explode` operator is similar to `map` in its handling of types: types may be left unspecified, and can be manually inferred by the developer before invoking `explode`. The resulting dataframe's types may be left unspecified or may be provided as an optional parameter to the `explode` operator.

`window` does not require unspecified types to be inferred and the resulting dataframe's types must match the input dataframe's data types.

The `window_reduction` operator does not force the materialization of unspecified types, but the resulting dataframe's types may be left unspecified, denoted by an optional `result_schema` parameter.

The `groupby` operator does not force the materialization of unspecified types, but the types of the resulting dataframe may be left unspecified if not provided via the `result_schema` parameter.

`reduction` does not force the materialization of the types, and the resulting dataframe's types may be provided by the developer via the `result_schema` parameter or left unspecified.

The `infer_types` operator by definition will infer the types of the columns selected, and the resulting dataframe's schema will reflect the type inference and coercion in this operators definition.

`join` will force the inference of unspecified data types when invoked, but only for the columns involved in the join. The output data types are not transformed from the two input dataframes' schemata.

The `concat` operator will not force type inference, and if types are already specified in the schemas, they can be inherited from the source dataframes.

`transpose` does not materialize the types, and the resulting dataframe's types are left unspecified after a transpose absent a homogeneous dataframe.

The `to_labels` operator will force type inference of the columns that will become the labels in order to to build an indexing structure for fast `mask` operations. This indexing structure is not necessarily needed immediately so this type inference operation can be deferred until the index is built on the new row labels.

`from_labels` will not force type inference for the resulting dataframes, whether the types of the labels are known or not. As we mentioned in the description of `from_labels` the types of the labels may not be known if no indexing structure has been built against the labels, and they are not forced into being inferred by this operator.

Logical Order Management

Logical order and position management is a key requirement of dataframe system. The system must be able to track the user's order (or directly tie the physical representation to the logical order) to correctly execute dataframe queries. We now go through each operator listed above and describe the logical order management rules.

In `mask`, the materialization of the positional notation values is forced in cases that the positional notation is used. The order of the resulting dataframe matches the order of the input labels, which need not match the input dataframe's order. An implementation could, for example, separate mechanisms for reorder and indexing, so if labels are received in a different order than the input dataframe's order, the labels are first extracted in order and the reorder is deferred to optimize performance.

For a `map`, the order of the columns (rows if `axis=1`) must be materialized before `map` in cases where the user-defined function is relying on the logical order. In practice, an implementation could materialize the column order when a `map` is invoked in cases where it is unclear or challenging to inspect the user-defined function and determine whether order is necessary in the user's function. The output order of the dataframe must be the same in both axes, with new columns (rows if `axis=1`) being added to the logical end.

In `filter`, the order of the rows (columns if `axis=1`) does not need to be materialized, but the order of the columns (rows if `axis=1`) must be materialized if the condition function provided by the developer relies in any way on the order of the columns. Neither the columns nor the rows are reordered by the `filter` operation.

In `explode`, the order of the columns must be induced in the case that the user-defined function argument relies on the ordering of the columns (rows if `axis=1`), but row (column if `axis=1`) order need not be materialized prior to the invocation of `explode`. The `explode` operator does not directly transform the order of the resulting dataframe with respect to the input dataframe, but the positional notation values will be updated as a result of an `explode`. The positional notation itself does not need to be immediately materialized after the operator has completed execution, but metadata is maintained to ensure that the order and positional notation values can be derived as they are needed. It takes a non-trivial amount of communication to determine the correct logical order and the precise positional notation if order was not previously materialized, so in these cases an implementation can defer the precise positional notation calculation until it is needed.

Because sliding window operations directly rely on the order of the data, the order must be materialized prior to executing the `window` operator. The logical order of the resulting dataframe will be the same as the input dataframe's logical order. For `window_reduction` similarly, the resulting logical order will remain the same as the input order, with the aggregations of each window in their logical locations with respect to the input dataframe.

`groupby` is value-based, so it does not rely on order. The logical order does not need to be materialized before the `groupby` is performed. The logical order of the resulting dataframe is the lexicographical order of the values in the `by` columns. The unique combination of values in the `by` columns become the row (column if `axis=1`) labels, and in the case of multiple `by` columns, a hierarchical label structure is created. The number of output columns (rows) is identical to the number of input columns (rows).

In a reduction, the logical order of the rows (columns if `axis=1`) must be materialized before the user-defined aggregation function can be applied in the case that it relies on the order of the data. The logical order of the columns (rows if `axis=1`) does not need to be materialized for the execution of the `reduction` operator, but the resulting dataframe's logical column (row if `axis=1`) order will be inherited from the input dataframe. The result's column labels (row labels if `axis=1`) are unchanged from the input dataframe, and the resulting row label (column label if `axis=1`) is set to the positional notation, 0 in this case since a `reduction` will always result in a single row.

`infer_types` does not interact with the logical order in any way, so the resulting dataframe's order does not need to be materialized and remains unchanged from the input dataframe.

In a `join` operation, the order of the resulting dataframe is determined from the two input dataframes, so the logical order of these dataframes must be materialized for the `join` to take place. The output dataframe is ordered by the logical order of occurrence of the left, followed by the logical order of the right.

For the `concat` operator, the order of the resulting dataframe is inherited from the left dataframe first, then in the order they occur in the array of the `others` parameter. The logical order of each input dataframe needs to be materialized before this operator can be applied.

The `transpose` operator does not rely on order, so the logical order does not need to be materialized. The logical order is, however, inherited from the input dataframe.

The `to_labels` operator does not rely on order, so the logical order does not need to be materialized. The logical order is, however, inherited from the input dataframe.

The resulting dataframe's types are inherited from the input dataframe. The order is neither materialized nor updated as a result of the `from_labels` operator.

4.5 Discussion

In this chapter, we presented our work on an architecture design for dataframes, such that other implementations may learn and borrow from this work. We presented a set of rules and optional design decisions to try to account for the more challenging components of the dataframe data model. Additionally, we discussed ways that relational database systems can incorporate and implement a subset of the dataframe data model in the architecture presented here. In the next chapter, we present our reference implementation of this design, Modin, and several of the low level design and implementation decisions we made to scale dataframes.

Chapter 5

Modin: The Reference Dataframe Implementation

5.1 Introduction

It is well-known that dataframe systems like pandas are non-interactive on moderate-to-large datasets, and break down completely when operating on datasets beyond main memory [87, 102, 55, 59, 60, 107, 114]. These issues represent significant challenges for users who are unwilling or unable to switch to other, more scalable tools, such as relational databases. To address these shortcomings, we have been developing Modin, a *parallel dataframe system*, acting as a drop-in replacement for pandas, over the past few years. Modin is already being used by data scientists in the telecommunications, finance, and the automotive industries, has been *downloaded more than 1 Million times*, with over 75 contributors across 12+ institutions, and more than 6000 GitHub stars (as of April 2021), demonstrating the value of such a system to the data science community. Modin's source code is open-source at `https://github.com/modin-project/modin`, with user and developer documentation at `https://modin.readthedocs.io/`. To build Modin, we had to address the dual problems of ensuring *scalability* of the rich set of dataframe operators when operating on the tolerant data model, while also providing clear, consistent, and correct *semantics* to users. In this chapter we operationalize and extend the dataframe algebra presented in Chapter 3 and the design in Chapter 4 in a real implementation: Modin. We primarily target two key aspects, each with their associated challenges:

Rule-based Decomposition. Unlike relational operators, dataframe operations can be carried out at the granularity of rows, columns, or even cells. For example, `fillna` accepts an input `axis` argument that specifies whether NULL values are filled along rows or columns. To apply dataframe operations in parallel, along rows or columns or cells, we must develop a set of *decomposition rules* that allow us to flexibly rewrite dataframe operations on the original dataframe into analogous operations on vertical, horizontal, or block-based partitions of the dataframe. Unlike in the relational context, these decompositions must preserve and maintain ordering. Further, they must be tolerant to the fact that the column types may change in the decomposed dataframes in unpredictable ways, requiring possible coordination across the decompositions, which is expensive. Moreover, the flexible data model blurs the boundary between data and metadata, and supports operators that query and manipulate the data and the metadata at the same time—identifying decomposition rules for parallelizing such operations is non-trivial. For example, `pivot` elevates a column of data into column labels (i.e., metadata used to identify columns) and reshapes the dataframe based on these new labels. Finally, we need to outline these decomposition rules for a core set of dataframe algebraic operators, with the understanding that the entire set of operations (in systems like pandas) can be rewritten using this core set. For this purpose, we draw on our proposed candidate algebra from Chapter 3, but extend it somewhat to make it more practically suitable—for example, recognizing that the prior algebra required us to repeatedly take transposes to apply columnar operations, we instead natively support columnar versions of operations. Irrespective, distilling the 600+ functions in a system such as pandas into a small core set of operators still poses a substantial engineering challenge.

Metadata Independence. Dataframe systems such as pandas make several metadata-related design decisions that both impact scalability and semantics. In particular, they tightly couple metadata with the physical representation of the dataframe; instead, we strive for *metadata independence*, where the metadata is captured at a logical level, with the physical representation of the metadata being decoupled from the logical. For instance, dataframe systems such as pandas eagerly determine and materialize the type of each column at the end of each operation—a time-consuming blocking step that severely slows down operations on large dataframes. Morover, pandas often coerces types when this may not be intended by the user, such as casting integers into floats in columns with a mix of both. Instead, our goal is to develop an *independent type system for dataframes* that natively supports mixed types and unspecified types in a column, whereby we can defer type inference to only when it is needed. Determining which algebraic operators require type inference is not straightforward. Another important design decision in present-day dataframe systems is to physically store data in logical order of rows and columns. While this is convenient in terms of accessing data by row or column number, it also eliminates a degree of freedom in terms of storage, and requires coordination after each operation to materialize the ordering information associated with each row and column. Instead, we need to support *order independence* wherein the physical order can be made to match the logical order on demand, but isn't done unless necessary. Overall, ensuring correct type and ordering semantics for the massive set of dataframe operators is a big challenge.

Our Approach. In this work, we address the scalability and semantics challenges and instantiate our ideas in Modin, our first cut at a parallel dataframe system. In Chapter 4, we formally described a small set of core operators with formalized type and ordering semantics that we implement within Modin. Our core operators include relational operators adapted to the dataframe context (e.g., ordered versions of `select`, `project`, and `join`), operators used to query and manipulate the metadata (e.g., converting data to labels), and low-level operators that enable the application of system or user-defined functions (e.g., `map` and `reduce`). To allow these operators to be performed in parallel at scale, we identify flexible equivalence rules that express each operator on the dataframe as operators on decompositions or partitions thereof, with a suitable ordered concatenation operator to "reassemble" the overall dataframe if needed. We formally describe the semantics of decomposition at various granularities. Modin internally uses these decomposition rules to rewrite computation, by employing a flexible partitioning scheme along rows, columns, cells, or blocks of cells, as necessary. We identify two types of optimization opportunities for significantly improving the system performance by intelligently applying the decomposition rules. We also propose a *dataframe type system* as implemented in Modin and describe how typing is inherited across the core operators, and develop techniques to support label- and order-based access without requiring the physical order to match the logical order. Overall, Modin provides up to a $100\times$ speedup relative to pandas and Koalas on a range of workloads including joins, type inference, and row-oriented UDFs.

Related Work. Recent efforts from the database research community has described how to rewrite dataframe operations into SQL [55, 60, 107]; while these efforts are valuable, they only rewrite a subset of the pandas API that is expressible as relational operators, leaving the rest to be executed as is in pandas. However, they do not natively support the dataframe data model (which supports mixed types, and row/column equivalence) or the vast majority of operations, which are non-relational (e.g., columnar operations, non-atomic operations, operations that move data to metadata and vice-versa). We describe other differences with respect to metadata management in Section 5.5. Koalas [63], Dask [31], and Ibis [53] are other dataframe implementations which support simple parallelization for row-oriented operations; however, as we will show in our experiments, they are unable to support columnar operations, or move data to metadata and vice-versa. Our decomposition or partitioning schemes (row-, column-, and block-wise partitioning) are analogous to matrix partitioning [40]; however, the matrix data model (with homogenous data types) and set of operators are both very different, necessitating different decomposition rules.

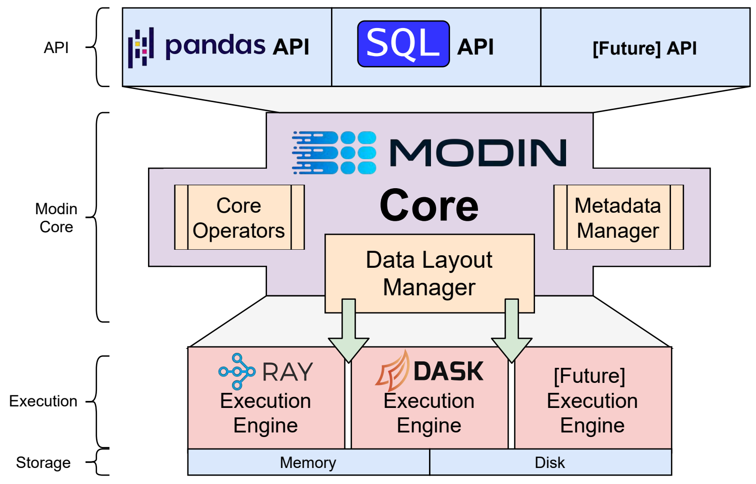


Figure 5.1: The Modin architecture

5.2 Pluggable Interaction Modalities

Modin was architected to address the scalability challenges with dataframes while also abstracting away system details that data scientists do not need to care about. In this section, we detail the components of the Modin architecture (shown in Figure 5.1) and present the challenges we address.

Modin's API layer is modular in order to support multiple modes of interaction (e.g., pandas API, SQL, or Apache Spark DataFrame API [109]). Supporting multiple modes of interaction is important to ensuring productivity because data scientists often feel more comfortable performing certain tasks in one language over another. For example, a data scientist may feel more comfortable writing a `JOIN` in SQL than pandas. In fact, there are many questions on StackOverflow [110] that ask "How do I write X in pandas/SQL?" where X is a SQL or pandas function respectively. To support the multiple modes of interaction, as well as provide the ability to extend to new ones, we define a compact set of operators that can be reused to implement existing APIs and define new ones.

Currently, Modin exposes the full pandas API (with 600+ functions) while many popular dataframe systems, such as Koalas [63] and Dask Dataframe [30], only support a subset of the pandas API that more directly corresponds to relational operators. Modin provides the pandas API as a drop-in replacement as-is, without changing its semantics, through the `modin.pandas` module. Even though there are many challenges associated with exposing the pandas API, we found that it is vital to support it exactly as is to include edge case behaviors, inconsistencies, and error messages—to avoid deviating from expected behavior for data scientists.

5.3 Modin Core

The Modin Core is the narrow-waist of Modin's architecture. It contains a set of core operators, decides the best data layout or partitioning to parallelize the core operators, and efficiently manages metadata. This design is intended to allow all user-level APIs to leverage the same performance optimizations. We now describe the structure of the Modin Core and defer the details of the operator parallelization and metadata sub-components to later sections.

Core operators and data layout manager

To allow Modin's core operators to be applied to large dataframes, Modin decomposes the dataframes into smaller partitions, enabling parallel execution of the operators on the partitions. However, the breadth of access patterns assumed by a given operator makes decomposition a challenge: operators that assume access to an entire row may be followed by operators that assume access to an entire column. Modin's physical layout can represent data as row-wise, column-wise, or block-wise partitioning transparently. Modin's data layout manager efficiently shuffles the data between these physical layouts to support challenging operator combinations. We discuss the decomposition rules of each operator and optimization opportunities that stem from applying the decomposition rules in Section 5.4.

Metadata manager

The metadata manager is responsible for maintaining the metadata associated with a dataframe, including data types, column and row labels, and the mapping between logical order and physical order. We give an overview of metadata management here and discuss the details in Section 5.5.

Data types. In dataframes, one column can contain values of one or more types. We develop a dataframe type system to formally define the semantics of querying and manipulating types in a column. Specifically, our type system organizes types into a tree hierarchy, where a parent node represents a more general type than its children. We also define how input types of a dataframe are transformed to output types for each core operator such that we have a clear semantics of transforming mixed types across a chain of operators. Since the cost of inferring types is high and the data types are query-able in dataframes, we additionally perform a few optimizations to improve Modin's performance, including delayed type inference and building indexes on data types.

Column and row labels. One of the biggest metadata challenges in dataframes is that metadata can become data, and vice-versa. For example, it is possible to insert the row labels into the data and operate on them as data. In addition to this interchange, users have expectations for low latency interactions when they lookup rows or columns by labels. Modin indexes the label metadata and move labels close to data to accelerate querying and manipulating labels.

Logical order. Modin's data layout decouples the logical order, also known as the user's order, from the physical order through indirection. Though the order of the data is known, the precise position of each row (i.e., row number) may not necessarily be known at any given time. For example, for `filter`, the order is unchanged, but the position of a given row may change. Since recomputing the specific position is costly and the specific position is rarely used, Modin will defer calculating position until necessary.

5.4 Modin Operators and Optimization

Modin supports a small set of core dataframe operators to implement the user-level APIs. This design requires addressing two major challenges. First, these operators need to be powerful and extensible such that they can be used to quickly implement new APIs or extend existing ones. Second, we need to identify decomposition rules for each operator such that the operator can be executed in parallel to reduce interactive latency. Modin addresses the first challenge by carefully choosing the set of core operators. First, we include dataframe versions of relational operations (e.g., `join`) since they are widely used in data analysis and are the building blocks for many user-level APIs. Second, we include non-relational operators that query and manipulate metadata (e.g., `infer_types` and `transpose`) to support flexible schema and mixed types. Finally, we include low-level operators (e.g., `map`, `groupby`, and `explode`) that accept an input function. The input function can be written by the user, e.g., the `apply` function in pandas which accepts a general purpose Python function as input, in which case we call this a *user-defined function (UDF)*. Or this function can be in-built into the system by the developer implementing the API in Modin (i.e., the authors and contributors to the Modin codebase), e.g., `fillna` in pandas, where `NULL` values are filled in using a specific approach. We call this a *system predefined function (SPF)*.

Modin addresses the scalability challenges by parallelizing the aforementioned core operators. We formally define the semantics of dataframe decompositions and propose a set of decomposition rules for parallelizing operators over dataframe decompositions. We show that applying the decomposition rules can significantly improve the performance of Modin in Chapter 6. More importantly, these rules are independent of the underlying execution engines (e.g. Ray or Dask) and applicable to new execution engines.

We start by briefly revisiting the dataframe data model we presented in Chapter 3 and formally define semantics of decomposing a dataframe in Section 5.4. Subsequently,

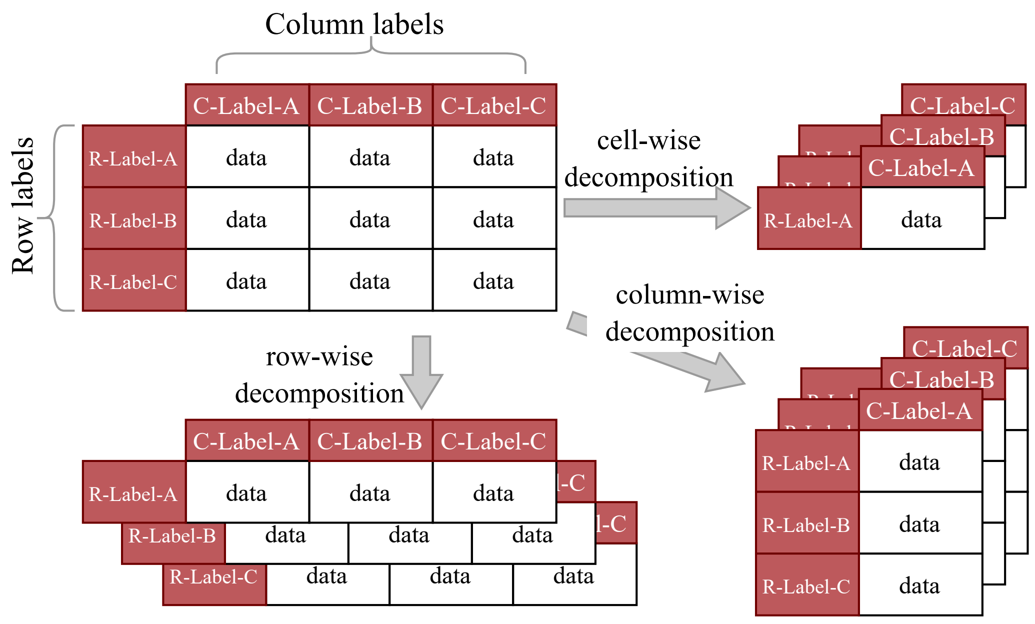


Figure 5.2: Cell/row/column-wise decomposition

we describe each operator and associated decomposition rules in Section 5.4. Finally, we illustrate additional optimization opportunities that stem from employing different decomposition rules for the same set of operators in Section 5.4.

Semantics of Decomposing a Dataframe

A dataframe D is defined as a tuple (A, R, C, T), where A is an $m \times n$ array of entries that represents the dataframe content, R is a vector of m row labels, C is a vector of n column labels, and T is the type information for each column, as we discussed in Chapter 3. The dataframe entries A are ordered. The operators that process a dataframe can either maintain the same order or modify it based the semantics of the operator, which will be described in more detail in Section 5.5. The row labels R and column labels C can be used to identify the corresponding rows and columns, respectively, and they do not have to be unique. Users can also use row/column numbers or positions to uniquely identify a specific row/column. A row/column number represents the natural order of the rows/columns of a dataframe (e.g., row number 1 represents the first row).

Decomposing a dataframe means dividing the dataframe content A into non-overlapping partitions, where for each partition A_k, we logically instantiate a new dataframe by adding the corresponding row labels R_k, column labels C_k, and type information T_k.

We propose five types of decompositions: `cell-wise`, `row-wise`, `column-wise`, `rowGroup-wise`, and `rowOrderGroup-wise`. Figure 5.2 shows the first three types. The cell-wise decomposition decomposes a dataframe into a set of `unit dataframes`. A unit dataframe $D_{ij} = (A_{ij}, R_i, C_j, T_j)$ includes a single value along with the corresponding metadata information. The row-wise and column-wise decomposition decomposes a dataframe into a set of row and column dataframes, respectively. A row dataframe $D_{i*} = (A_{i*}, R_i, C, T)$ means appending all of the unit dataframes with the same row labels as new columns in order, maintaining the natural order of the dataframe D. We denote this append operation as $\bigoplus_c$.

$$D_{i*} = \bigoplus_{j=1}^{n}{}_{c} D_{ij}$$

$\bigoplus_c$ can be generalized to append any dataframes with the same row labels and therefore the same number of rows. $\bigoplus_r$ is analogously defined as appending dataframes with the same column labels as new rows. Therefore, a column dataframe means appending the unit dataframes with the same column label as new rows. Note that unlike the relational context where we simply union horizontal partitions of a relation, here, special care must be taken to preserve the ordering of the dataframe partitions (which are themselves ordered) along rows and columns. The three types of decomposition, as in Figure 5.2, can be summarized as follows:

$$\begin{aligned} D &= \bigoplus_{i=1}^{m}{}_{r} \bigoplus_{j=1}^{n}{}_{c} D_{ij} = \bigoplus{}_{*} D_{ij} \\ &= \bigoplus_{i=1}^{m}{}_{r} D_{i*} \\ &= \bigoplus_{j=1}^{n}{}_{c} D_{*j} \end{aligned}$$

The first line represents *cell-wise decomposition*, for which we use $\bigoplus_*$ as shorthand. The second and the third line represent row-wise and column-wise decomposition, respectively.

The rowGroup-wise decomposition is a special case of row-wise decomposition, where we partition the dataframe into groups of rows based on a composite key of a set of columns $cols$ and each group i includes the rows whose composite key equals a distinct key k_i. The rowGroup-wise decomposition can be represented as

$$D = \bigoplus_{i=1}^{l}{}_{g(cols)} D_{k_i}, \text{ where } D_{k_i} = filter_r(D, cols = k_i)$$

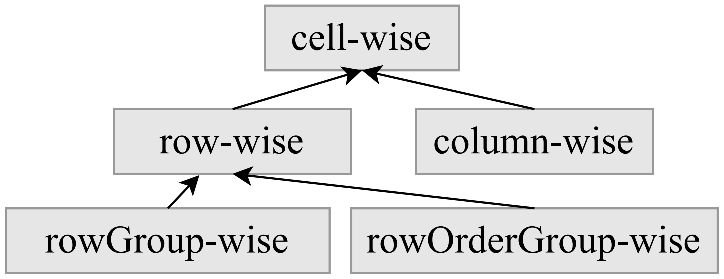


Figure 5.3: The hierarchy of decompositions: a parent node represents a more general decomposition than its children.

$filter_r$ means selecting the rows whose $cols$'s composite key equals k_i and $\bigoplus_{g(cols)}$ means that the groups are appended in the natural order that they arise in the dataframe. This decomposition is commonly used in operators such as group-by and equi-join. Another decomposition is the rowOrderGroup-wise decomposition. Compared to rowGroup, which uses the natural order, rowOrderGroup orders the groups by the groupby key, which is used by the sort operator. We will discuss this decomposition in Section 5.4 when we introduce the sort operator.

Decomposition Rules for Operators

We now describe the core operators in Modin and their associated decomposition rules. Each decomposition rule uses one or more types of decompositions discussed above. The five types of decomposition form a tree structure (shown in Figure 5.3) where a parent node represents a more general decomposition than its child nodes. For example, a row-wise decomposition can be viewed to be a cell-wise decomposition, but not the other way around. In addition, since a rowGroup-wise decomposition partitions a dataframe into groups of rows, it is a special case of the row-wise decomposition. When discussing the decomposition rules of each operator, we use the most general possible types of decomposition because replacing this decomposition type by its descendants will also result in valid decomposition rules for this operator. Note that if an operator processes the input dataframe at the granularity of rows/columns, we say that it is operating along the row/column *axis*, respectively.

We first discuss the low-level operators. Then, we present the decomposition rules for non-relational operators that query and manipulate metadata. Subsequently, we discuss the operators adapted from relational operators. We defer discussion on metadata, like type inference and ordering, to Section 5.5. In the following, we use f to represent a UDF or SPF (system predefined function) while h is used to represent an SPF.

Rulebox 1: decomposition rules for low-level operators

`map` : $map_*(f_*^{map}, D) = \bigoplus_{i=1}^{m}{}_r \bigoplus_{j=1}^{n}{}_c f_*^{map}(D_{ij})$

`explode` : $explode_r(f_r^{exp}, D) = \bigoplus_{i=1}^{m}{}_r f_r^{exp}(D_{i*})$

`groupby` : $gb(op, param, cols, D) = \bigoplus_{i=1}^{l}{}_{g(cols)} op(param, D_{k_i})$

where $D_{k_i} = filter_r(cols = k_i, D)$

`reduce` : $reduce_r(f_r^{red}, D) = \bigoplus_{i=1}^{m}{}_r f_r^{red}(D_{i*})$

Low-level operators

The low-level operators include `map`, `explode`, `groupby`, and `reduce`.

`map` and `explode`: The `map` operator accepts a UDF or SPF to transform an input dataframe into a new dataframe maintaining the same shape and metadata (e.g., row/column labels) as the input. If the UDF/SPF f_*^{map} is applied to each cell and outputs a single value, the `map` operator can use cell-wise decomposition map_* as shown in Rulebox 1. Based on Figure 5.3, `map` also supports the descendant decompositions (e.g., a row-wise decomposition, map_r, is also possible if f is applied to each row). One example usage of `map` is to implement `fillna` that fills `NULL` values using a specified method.

The `explode` operator uses a UDF/SPF to transform an input dataframe into a new one with a different shape and metadata from the input. The SPF/UDF can be applied row-wise or column-wise. When applied row-wise (i.e., f_r^{exp} in Rulebox 1), each row expands into one or more rows, while maintaining the same column labels. Similarly, f_c^{exp} can transform a column into one or multiple columns with the same row labels. When new rows or columns are generated, their corresponding row or column labels are derived from the input counterparts. Therefore, the `explode` operator supports row-wise (i.e., $explode_r$ in Rulebox 1) and column-wise decompositions, depending on how it is applied.

`groupby`: As shown in Rulebox 1, the `groupby` operator takes a dataframe D, a set of groupby columns $cols$, and a Modin operator *op* with parameters $param$ as input. It groups the rows of the dataframe based on the composite key of the groupby columns $cols$, and applies the input Modin operator *op* to each group[1], thereby supporting the rowGroup-wise decomposition. One example usage is to replace `NULL` values in each group with a value that is based on the key of the groupby columns $cols$. In this case, a `map` can be used to replace `NULL` values for each group.

[1]Currently, Modin does not allow operators that change the number of columns or the column labels in a `groupby` operator

Rulebox 2: decomposition rules for metadata operators

$$
\begin{aligned}
\texttt{inferT} &: inferT(D) = \bigoplus_{j=1}^{n}{}_{c}\, h_c^{infer}(D_{*j}) \\
\texttt{filterT} &: filterT(D,t) = \bigoplus_{i=1}^{m}{}_{r} \bigoplus_{j=1}^{n}{}_{c}\, mask(h_*^{lb}(t,D), D_{ij}) \\
\texttt{to\_labels} &: to\_labels(cols, D) = \bigoplus_{i=1}^{m}{}_{r}\, h_r^{to}(cols, D_{i*}) \\
\texttt{from\_labels} &: from\_labels(D) = \bigoplus_{i=1}^{m}{}_{r}\, h_r^{from}(D_{i*}) \\
\texttt{transpose} &: transpose_*(D) = \bigoplus_{i=1}^{m}{}_{r} \bigoplus_{j=1}^{n}{}_{c}\, h_*^{trans}(D_{ij})
\end{aligned}
$$

`reduce`: The `reduce` operator aggregates each row/column dataframe into a single value based on a SPF/UDF (e.g., f_r^{red} in Rulebox 1); one possible SPF could be `average`. Therefore, the row-wise decomposition (i.e., $reduce_r$ in Rulebox 1) breaks the dataframe into row dataframes D_{i*}, applies the function f_r^{red} to each one, and outputs a unit dataframe. We note that for some functions (e.g., `sum`), one possible optimization is to further decompose a row dataframe D_{i*} into smaller partitions, apply this function for each partition, and aggregate the results. The column-wise decomposition of `reduce` is defined symmetrically.

Operators for querying and manipulating metadata

The operators for querying and manipulating metadata include `infer_types`, `filter_by_types`, `to_labels`, `from_labels`, and `transpose`.

`infer_types` and `filter_by_types`: Since Modin supports mixed types in a column, we provide the `infer_types` operator to infer the type of a column by inspecting the type of each cell within the column and finding the common type. Modin organizes the types in a tree structure, where a parent node represents a more generic type than its child nodes. Section 5.5 introduces a dataframe type system, as implemented in Modin. The `infer_types` operator applies a SPF h_c^{infer} to each column dataframe and generates a new one with the updated type information (rule `inferT` in in Rulebox 2). The `filter_by_types` operator checks the column types and filters out the columns whose types are not in a specified list of types (rule `filterT` in Rulebox 2). It uses a SPF h_*^{lb} to find the column labels whose column types are in the specified types t and adopts a `mask` operator to project the corresponding columns. The `mask` operator extracts cells based on the specified row/column labels and will be discussed in Section 5.4.

`to_labels` and `from_labels`: `to_labels` replaces the dataframe's row labels with one or more columns of data, while `from_labels` operator converts the row labels into a column. Both operators support row-wise decomposition, but not column-wise. Their

Rulebox 3: decomposition rules for relational operators

$$\begin{array}{rl}
\texttt{mask} & : mask_*(labels, D) = \bigoplus_{i=1}^{m}{}_r \bigoplus_{j=1}^{n}{}_c h_*^{mask}(labels, D_{ij}) \\
 & : mask_r(rnSet, D) = \bigoplus_{i=1}^{m}{}_r \mathbb{I}[i \in rnSet] D_{i*} \\
\texttt{filter} & : filter_r(f_r^{flt}, D) = \bigoplus_{i=1}^{m}{}_r f_r^{flt}(D_{i*}) \\
\texttt{window} & : window_r(f_r^{win}, w, D) = \bigoplus_{i=1}^{m}{}_r \bigoplus_{j=1}^{n}{}_c f_r^{win}(\bigoplus_{k=j}^{j+w}{}_c D_{ik}) \\
\texttt{sort} & : sort(cols, D) = \bigoplus_{i=1}^{m}{}_{o(cols)} h_o^{sort}(cols, D_{[p_i, p_{i+1})}) \\
 & \text{where } D_{[p_i, p_{i+1})} = filter_r(p_i \leqslant cols < p_{i+1}, D) \\
\texttt{join} & : join(cols^l, D^l, cols^r, D^r) = join(\bigoplus{}_g D_k^l, \bigoplus{}_g D_k^r) \\
 & \qquad = \bigoplus{}_g cross\_prod(D_k^l, D_k^r) \\
 & \text{where } D_k^l = filter_r(cols^l = k, D^l) \\
 & \qquad D_k^r = filter_r(cols^r = k, D^r) \\
\texttt{concat} & : concat_r^{out}(D^1, D^2) = \bigoplus_{k \in \{1,2\}}{}_r \bigoplus_{i=1}^{m_k}{}_r h_r^{out}(labels_{out}, D_{i*}^k) \\
 & : concat_r^{in}(D^1, D^2) = \bigoplus_{k \in \{1,2\}}{}_r \bigoplus_{i=1}^{m_k}{}_r mask_r(labels_{in}, D_{i*}^k)
\end{array}$$

decomposition rules are presented in Rulebox 2. `to_labels` uses the SPF h_r^{to} to replace each row dataframe's row label with the data in columns $cols$ and deletes the $cols$ to generate a new row dataframe. The new row dataframes are appended to generate the output. `from_labels` uses SPF h_r^{from} to do the opposite.

`transpose`: The `transpose` operator switches the row and column data of a dataframe. It supports cell-wise decomposition: for each unit dataframe, we swap the row and column label using a SPF h_*^{trans} as shown in Rulebox 2. We note that one system optimization in Modin is that we do not necessarily physically swap data and labels for the `transpose` operator, instead modifying the mapping from physical to logical for a no-shuffle dataframe transposition.

Relational operators

The dataframe operators that are adapted from relational operators include `mask`, `filter`, `window`, `sort`, `join`, `rename`, and `concat`.

`mask` and `filter`: The `mask` and `filter` operators are adapted from relational operators `project` and `select`. The main difference from their relational counterparts is that `mask` and `filter` can be applied to both the row and column axes, and the output

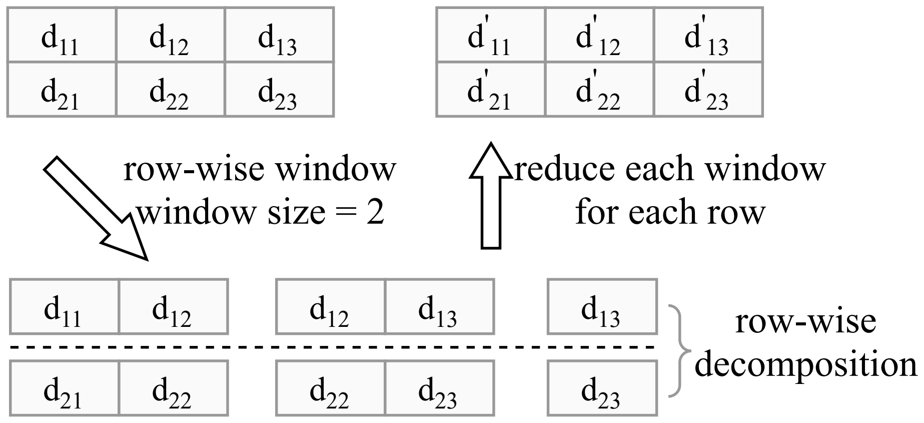

d_{11} d_{12} d_{13}
d_{21} d_{22} d_{23}
d'_{11} d'_{12} d'_{13}
d'_{21} d'_{22} d'_{23}
row-wise window
window size = 2
reduce each window
for each row
row-wise
decomposition

Figure 5.4: An example of window operator

dataframe maintains the same ordering as the input. The `mask` operator allows developers to project and select the entries in a dataframe using column labels and row labels together. `mask` also allows developers to specify the row and column numbers. A `mask` that subselects dataframe entries based on labels supports cell-wise decomposition, that is, for each unit dataframe, the `mask` discards this unit dataframe if its corresponding row and column labels are not in the specified labels. Similarly, a `mask` that subselects dataframe entries by specified row numbers also supports cell-wise decomposition, where unit dataframes are discarded if their row number is not in the specified set. We express this using an indicator function $\mathbb{I}[i \in rnSet]$ in Rulebox 3. The column case is symmetric. The `filter` operator eliminates rows/columns that do not satisfy certain data-specific conditions (as opposed to label/order-specific conditions as in `mask`) as encapsulated in a SPF/UDF. Rulebox 3 shows the decomposition rules for `mask` and `filter`.

`window`: The `window` operator performs a sliding window operation by grouping dataframe cells in a column-wise or row-wise manner, and for each set of windowed cells, uses a SPF/UDF to `reduce` them to a single value. We use an example in Figure 5.4 to explain the decomposition rule of `window` in Rulebox 3. Here, the window size is 2 and the `window` operator operates on the row axis. So we use row-wise decomposition and for each row dataframe, we perform a window operation (i.e., each window includes 2 cells or less shown in Figure 5.4). For each window of cells (i.e., $\bigoplus_{k=j}^{j+w}{}_{c}\, D_{ik}$ in Rulebox 3), we use a function f_r^{win} to reduce them into a unit dataframe. The generated unit dataframes are appended as new columns to generate a new row dataframe (via $\bigoplus_{j=1}^{n}{}_{c}$). Finally, the row dataframes are appended as new rows. The column-wise decomposition can be defined symmetrically and is omitted.

sort, join, rename, and concat: The sort and join operators have the same semantics as the relational counterparts. Their decomposition rules are shown in Rulebox 3. The sort operator uses rowOrderGroup-wise decomposition (i.e., $\bigoplus_{o(cols)}$), where dataframe rows are range-partitioned based on the sorting columns $cols$ such that the $cols$ values across partitions are ordered. As shown in Rulebox 3, the $cols$ values of the rows in one partition i fall into a value range $[p_i, p_{i+1})$, where p_i is the minimum key of a partition. We then use the function h_o^{sort} to sort each partition independently to complete the sort operation. The join[2] operator supports rowGroup-wise decomposition. The rows of input dataframes are partitioned by the join keys (i.e., $cols^l$ and $cols^r$ for D^l and D^r in Rulebox 3, respectively) and each pair of partitions D_k^l and D_k^r is joined locally using cross product $cross\_prod$. The rename operator replaces the input dataframe's row and column labels with the specified new labels. Since rename does not access the dataframe content, it does not have a decomposition rule.

The concat operator is analogous to union in relational algebra. The difference here is that concat does not require the input dataframes have the same row or column labels and can applied on both the column and row axes. Additionally, concat maintains the row and column ordering of the input dataframes. Our following discussion focuses on row-wise concat; here, concat appends rows while joining their column labels. Modin currently supports inner and outer label join. concat with outer label join (i.e., $concat^{out}$ in Rulebox 3) includes three steps: 1) take the union of the column labels of two input dataframes (i.e., $labels_{out}$); 2) for each row dataframe, use a function to extend its column labels to the union column labels and filling the newly generated cells with NULL (i.e., $h_r^{out}(labels_{out}, D_{i*}^k)$); 3) append the new rows together. concat using inner label join takes the intersection of the input column labels (i.e., $labels_{in}$) and uses the intersected column labels to project the input rows (i.e., using $mask_r(labels_{in}, D_{i*}^k)$).

Applying Different Decomposition Rules

We now identify two potential optimization opportunities made possible by intelligently choosing between decomposition rules. Since some operators can be decomposed in different ways, we can change the decomposition pattern based on the immediate preceding or succeeding operator decompositions. For example, a map_* operator can be rewritten to map_r or map_c and maintains the same semantics because map_* is a more general version of map_r and map_c as shown in Figure 5.3. Choosing different decomposition rules for the same set of operators can result in different performance. Our experiments in Section 5.4 demonstrate that selecting decomposition rules appropriately can significantly improve the performance of Modin.

[2]For simplicity, we assume an equi-join and omit other join types.

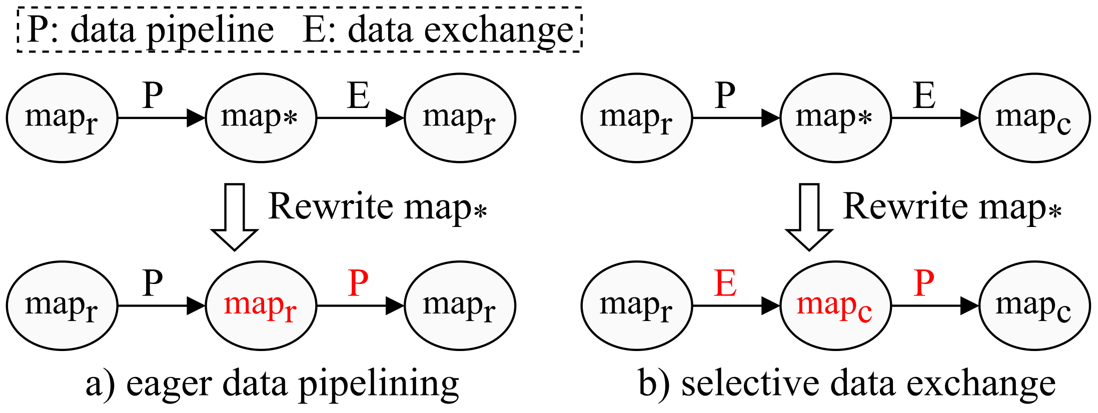
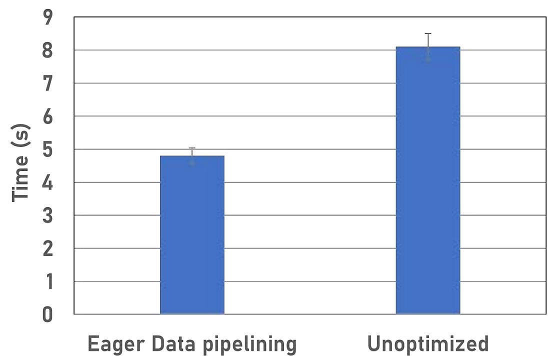


Figure 5.5: Optimization opportunities from applying different decomposition rules

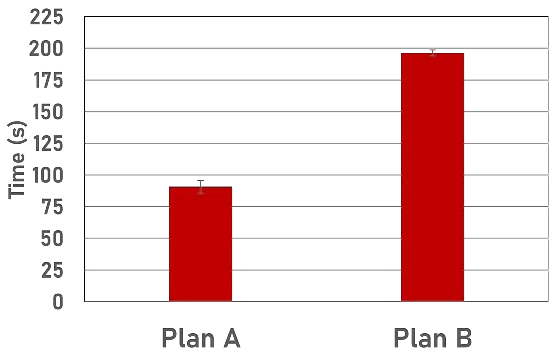
Eager data pipelining: This optimization applies the decomposition rules to allow more data pipelining. Figure 5.5(a) shows an example. Here, users issue three chained map operators. The SDF/UDFs of the first and the third operator need to be applied to each row (i.e. map_r) while the second SDF/UDF can be applied to each cell (i.e., map_*). Independently applying the decomposition rules for each operator results in a plan where the first and the third operator use row-wise decomposition and the second operator uses cell-wise decomposition. We can pipeline the data from a row-wise decomposition to a cell-wise decomposition, but need to exchange [42] data (via data shuffling) if the order of the two decompositions is reversed because the cell-wise decomposition is more general than row-wise decomposition. Therefore, the first plan in Figure 5.5(a) requires data exchange when we pass data from the second to the third operator. One optimization opportunity here is if we "downgrade" the cell-wise decomposition into a row-wise decomposition, then the three operators can be pipelined as shown in the second plan of Figure 5.5(a). Therefore, an interesting optimization here is how to opportunistically rewrite a decomposition into a more specific one to enable more data pipelining.

Selective data exchange: We can also apply the decomposition rules to swap data exchange and pipeline across different operators. Data exchange is generally more costly than data pipelining. Therefore, we prefer to exchange (or shuffle) less data at the cost of pipelining more data. Figure 5.5(b) shows an example where users issue three map operators. The first and third one require row-wise (i.e., map_r) and column-wise decomposition (i.e., map_c), respectively. The second one uses a cell-wise decomposition (i.e., map_*). In this plan, we need to exchange data between the second and the third operator. An alternative plan is to rewrite the cell-wise decomposition into a column-wise one (i.e., the second plan in Figure 5.5(b)). This plan needs to exchange data for the first two operators with the benefit of pipelining data between the second and the third operators. Depending on the

Figure 5.6: Performance impact of data pipelining and selective data exchange.



(a) Performance benefit of eager data pipelining



(b) Performance benefit of selective data exchange

amount of data passed across the three operators, the two plans prevail in different cases. The optimization here involves applying the decomposition rules to find the best plan that reduces the cost of data exchange.

Performance impact of choosing different decomposition rules

We now test the performance impact of choosing different decomposition rules and show the optimization opportunities from eager data pipelining and selective data exchange discussed.

We first explore the optimization opportunity from eager data pipelining. Recall that eager data pipelining pipelines operators that decompose cell-wise in between two operators that decompose row or column-wise. We test three `map` operators that are chained as $map_r \rightarrow map_* \rightarrow map_r$, where each map operator accepts a UDF that transforms `NULL` values in the dataset into a new value depending on the column type. map_r operates on each row and pipelines data to map_*, which operates on each cell. Since map_* is followed by map_r, it needs to do a data exchange. The eager data pipelining technique rewrites this query into $map_r \rightarrow map_r \rightarrow map_r$ because map_* is a more general decomposition than map_r. This way, we can pipeline the three operators. Figure 5.6a shows the execution time of the two plans. We observe that the execution time of the optimized plan is 57% of that of the original plan. The majority of the overall reduction in the execution time is due to reduced communication between operators.

The second technique we explore is selective data exchange. Selective data exchange can occur when an operator that decomposes cell-wise is surrounded by each of the other two decompositions: row and column, which is a common pattern in regular dataframe workloads. We test two plans that have equivalent semantics but different performance. The first plan is $map_r \rightarrow map_* \rightarrow map_c$ (denoted as `PlanA`), which includes a data pipelining for the first two operators and a data exchange for the last two. The first operator map_r outputs significantly more data than the second operator map_* because the first operator converts each input row from the NYC dataset to a row of strings while the second operator outputs the first character of each input string. The map_c operator converts strings to numbers if possible, otherwise leaves the value unchanged. An alternative plan is $map_r \rightarrow map_c \rightarrow map_c$ (denoted as `PlanB`), which enforces exchanging data first and then pipelining. We expect PlanB to be more costly because it exchanges more data than PlanA. Figure 5.6b shows the results of the two plans. We see that the two plans have very different execution times: the execution time of PlanA is 35% of PlanB. Therefore, our decomposition rules allow more optimization opportunities; how to optimally pick between decomposition rules is left for future work.

5.5 Metadata Management

Modin manages various types of dataframe metadata: data types, the row/column labels, and the mapping between the logical order of columns and rows to the physical order. We now discuss each of these components individually.

Data Types

Unlike relations, columns in a dataframe can have mixed types, which poses multiple unique challenges. Dataframes must correctly and efficiently decide the type of a column that includes data of multiple types, as well as define the semantics of how each operator modifies the type information. Existing dataframe systems do not have a systematic solution for supporting mixed types. For example, pandas casts all integers to floating point within a column of mixed floats and integers. In addition, pandas does not support `NULL` values in integer columns by default, instead casting those integers to floating point. Consequently, an outer, left, or right join with missing values can result in a modified data type in pandas. Additionally, pandas eagerly infers each column type after most operations, significantly increasing execution time since inferring a column's type requires inspecting every value in a column.

We propose a type system for dataframes to address this challenge. We organize types into a hierarchy, where a parent node is a more general type than its children. Then, we define how the core operators modify types. With the clear semantics defined, Modin can defer type inference to when it is absolutely necessary, thereby reducing the cost of managing mixed types.

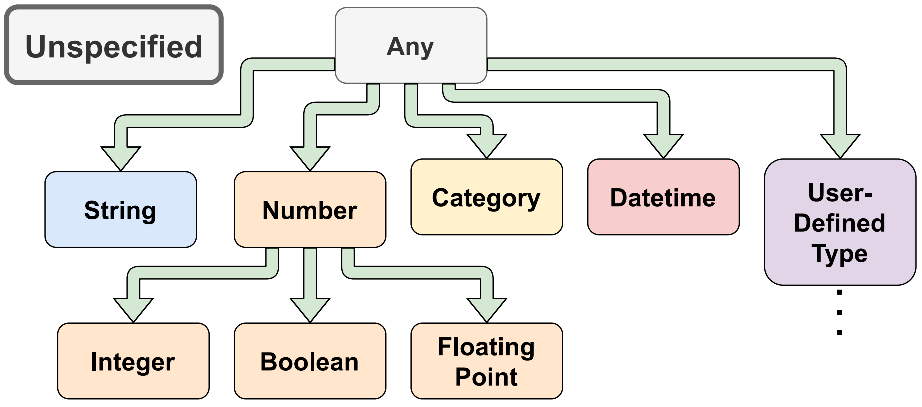


Figure 5.7: Dataframe Type System Hierarchy

Dataframe Type System. Our type system enables support for the unique properties of dataframes: mixed types in a column, unspecified types (i.e., the type information is only inferred when necessary), and type inference. Types are organized into a hierarchy and Figure 5.7 shows one instantiation. Here, types including integers, boolean, float are regarded as a number type. This number type along with string, category, and other types inherit `ANY`. We additionally have designation we call `UNSPECIFIED`, which represents columns where the type has not been determined yet. In Modin, all types, except `ANY` and `UNSPECIFIED`, are basic types and inherit `ANY`, but Modin can support more complicated types using the proposed type system. Our type system defines types only along columns and follows two invariants.

Invariant 5.5.1. *The output column types of the operators that accept a UDF/SDF is either provided at invocation or designated as* `UNSPECIFIED` *and implicitly inferred by the dataframe system. Type inference is deferred until an operator requires it.*

A column type with the designation of `UNSPECIFIED` can occur after operators that allow UDF/SDFs (e.g., `map`). This designation enables the user to apply functions anonymously without needing to know what the output type(s) will be, and allows performance optimizations that avoid calculating and materializing type information when it may never be needed by the user. We note here that `UNSPECIFIED` does not inherit `ANY`, because `UNSPECIFIED` is a designation specifically used to defer the materialization and inference of a given column's type.

Invariant 5.5.2. *A dataframe column i's type T_i is always correct, even though the type T_i may not represent the most precise type for that column i.*

Modin does not implicitly recalculate types that are already materialized, even if there is a more specific type that can describe a given column. Suppose a dataframe column has all integers except a single string, resulting in a column of type `ANY`. Here, a data scientist can remove the string with a `filter`, resulting in a column where all data values are integers. In this case, the type of the column remains `ANY`, despite a more precise type designation being possible. Modin can match the behavior of pandas by calling `infer_types` as a post-processing step to any pandas function. Our type system gives users the flexibility to choose to defer the type inference for performance, or to match the pandas semantics by calling `infer_types` after a given pandas function.

Type Rules by Operator. Each operator described in Section 5.4 has two rules for handling the column types: 1) whether the input types must be known to perform the operator, and 2) whether the output types are inherited from the input dataframe(s) or the output types may be specified or are `UNSPECIFIED`. Table 5.1 describes the data type handling rules for each operator. The column labeled "Input Types" describes whether the data types must be specified before that operator is applied. In the case of `sort` and `join`, the input dataframe types must be known upfront to determine whether or not the values can be compared. The type system will infer and update the types implicitly via `infer_types` if the input dataframe's types are `UNSPECIFIED` and the operator needs to know the input types (i.e., "Inferred" in Table 5.1). The "Output Types" column describes how the output types are derived. "Inherited" means that the output data types will match the input dataframe's types or remain `UNSPECIFIED`. For example, in the case of a `filter`, types are not modified, even if they are `UNSPECIFIED`. This presents an interesting opportunity where Modin can choose to maintain the lineage of dataframes throughout a workflow and propagate inferred type information to the related dataframes, so that the types only need to be calculated once based on this inheritance model. For operators that allow a UDF/SDF as input, the output types can be specified by the developer (i.e., "Specified" in Table 5.1), or left unspecified. Suppose, for example, a developer wants to perform a `map` with a SDF that returns `TRUE` for non-`NULL` values, and `FALSE` otherwise. Since all columns in the output are known to be `boolean`, this information can be provided by the developer implementing the SDF to the `map` upfront to avoid costly type inference. Alternatively, when the types of the output dataframe after a `map` is not known, it ends up being `UNSPECIFIED` for every column.

Another important thing to note is that Modin never implicitly coerces types. Whenever `result_schema` is passed as an argument to an operator, Modin infers the schema of the resultant dataframe, and checks whether it matches the provided `result_schema` - raising an exception if it does not. There is no native way to coerce types in Modin. Developers who wish to coerce types would need to call the `map` algebraic operator with a user defined function that encapsulated the coercion logic.

Table 5.1: Type inference and changes by operator.

Operator	Input Types	Output Types
`mask`	N	Inherited
`filter_by_types`	Y	Inherited
`map`	N	Specified or Unspecified
`filter`	N	Inherited
`explode`	N	Specified or Unspecified
`reduce`	N	Specified or Unspecified
`window`	N	Specified or Unspecified
`groupby`	N	Inherited
`infer_types`	N	Inferred
`join`	Y	Inherited
`concat`	N	Inherited
`transpose`	N	Unspecified
`to_labels`	N	Inherited
`from_labels`	N	Inherited
`sort`	Y	Inherited
`rename`	N	Inherited

Dataframe Label and Order Management

We now discuss how Modin manages labels and order.

Dataframe label management. The labels of a dataframe are part of the metadata, but have unique properties which allow them to be treated as data at any point. This presents an interesting challenge: the metadata manager must be flexible enough to allow the labels to move into the data (i.e., `to_label`) and vise versa (i.e., `from_label`). In addition to the flexibility of the labels, there are also latency expectations for `mask`. This presents the additional challenge that the system must be able to quickly execute queries on the labels, while also remaining flexible enough to move the labels into the data. Modin addresses this challenge by maintaining two sets of labels. One set of labels is placed near the data to allow fast conversion between labels and data, the other set is maintained externally as an indexing structure to support querying based on labels. Modin lazily synchronizes the two sets of labels when one set of labels is changed and the other set is accessed.

Another challenge of dataframe labels is support for duplicate labels. Since row labels can originate from one or more columns of data, and column labels can originate from row labels via a `transpose`, multiple labels with the same value are possible. Some operators like `join` and `concat` do not support duplicate labels in the column axis because there is no good way to define the correct behavior of these operators in the presence of duplicate columns.

Table 5.2: Order and position needs and changes by operator.

Operator	Input Order	Position	Output Order & Position
`mask`	N	Y*	Parameter-Dependent
`filter_by_types`	N	N	Inherited | Updated
`map`	N	Y⋄	Inherited from Inputs
`filter`	N	Y⋄	Inherited | Updated
`explode`	N	Y⋄	Inherited | Updated
`reduce`	N	Y⋄	Inherited
`window`	Y	N	Inherited
`groupby`	N	N	Data-dependent
`infer_types`	N	N	Inherited
`join`	N	N	Inherited*
`concat`	N	N	Inherited*
`transpose`	N	N	Inherited
`to_labels`	N	N	Inherited
`from_labels`	N	Y	Inherited
`sort`	N	N	Data-dependent
`rename`	N	N	Inherited

Logical Order management. Dataframes are logically ordered, typically with an order preferred by the user or inherited from data ingestion. The logical order provides an intuitive and consistent view of the data: after each transformation, the same rows/columns are shown in the same order. In addition, each row/column is associated with its numeric offset, or *position*, and users can select rows/columns based on this position via `mask`.

In dataframe systems like pandas, the logical and physical layer are tightly coupled, which can be beneficial at small scales, but quickly breaks down as datasets grow. Instead, we propose a logical order management system that maintains the logical order and physical positions separately, that is, Modin will maintain the logical order, but not materialize the position. Instead, the positions are only computed when requested. Materializing and maintaining positions is costly, and positions are not frequently used. For example, `filter` on the row axis does not change the order of the rows, but changes the positions of many rows. Maintaining the positions is costly since it requires a full scan of the dataframe. Our approach of lazily computing the positions avoids the maintenance cost and reduces the execution time of each operator. In practice, it is also much more likely that the logical order will be used than it is that the positional notation will be used, so tracking these separately has performance benefits to a typical workflow as well.

The rules for order and positions materialization and updates for each operator are listed in Table 5.2. The "Input Order" column specifies whether the column's order needs to be known (but not the position) before the operator can be applied. Among the operators, `window` is the only operator that requires order but not position information, because `window` parameter SDF/UDFs operate anonymously on the sliding window. The "Position" column specifies whether the specific positions must be computed before the operator can be applied. These are distinct requirements because there are cases where the order may be known implicitly but not the position. For `mask`, the positions are only needed when the parameters call for using position as the selection criteria; for label-based `mask`, positions are not required. The values marked with a Y$\diamond$ in the "Position" column only require the position to be materialized on the axis *opposite* that which the operators are applied. For example, to apply a `map` across the rows (map_r), the system need not calculate the positions for the rows because the operator is decomposed across that axis. In this case, the column positions are required on the input dataframe because the SDF/UDF can access values based on position. Operators that decompose cell-wise do not need to calculate the positions of the input dataframe. Consider a filter which removes rows: the rows will keep the same order, so it is possible to know the order. In this case, however, the position will be changed and it requires a non-trivial amount of computation to determine the new position. Internally, Modin tracks the position and the row order separately in order to avoid costly calculation of unnecessary metadata. In the case of `mask`, the position is needed to be known only if the `mask` parameters are for positional notation.

The last column of Table 5.2, "Output Order & Position" shows how the output order is determined. "Parameter-dependent" means the order and positions are updated based on the values provided to the operator as parameters. For `mask`, the order of the parameter labels (or positions) is the output order and positions are derived from these parameters. "Inherited | Updated" indicates that the output order is identical to the input dataframe's order, but the positions are changed (e.g., `filter`). "Inherited" means the order and positions remain unchanged from the input (e.g., `map`). "Data-dependent" indicates that the order and positions are derived from the data values, usually due to sorting or grouping. One example here is `groupby`, which groups rows/columns and generates a new order based how groups are generated and appended. The order of `join` and `concat` are based first on the order of the left input dataframe, then on the right input dataframes(s).

5.6 Partitioning

In this section, we describe the partitioning layer and how Modin handles and exposes partitioning. The unique properties of dataframes that make them easier for users to manipulate data, like transposability and being able to operate on either rows or columns arbitrarily, also make distributed implementations more challenging and complex, as we described in Chapter 3.

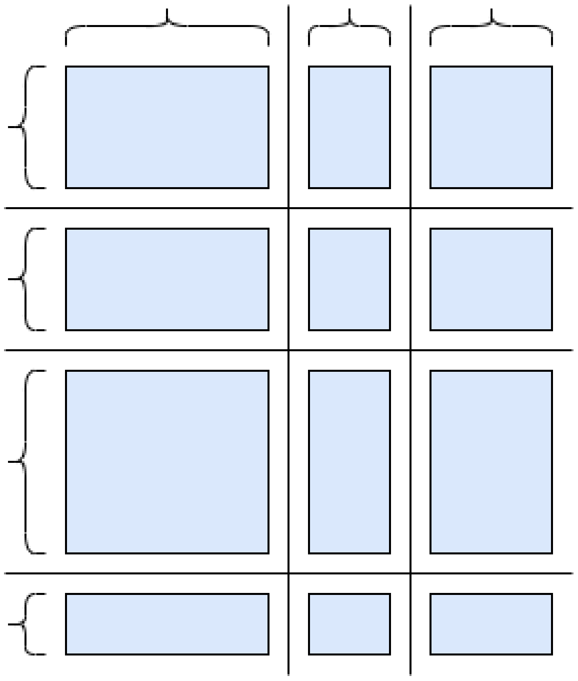


Figure 5.8: An example of valid partitioning layout in Modin.

Partition Layer

To solve the unique issues related to the interchangeability of columns and rows in dataframes, The Modin DataFrame uses a block partitioning schema, which partitions along both columns and rows. This partitioning approach gives Modin flexibility and scalability in both the number of columns and the number of rows supported. There is no minimum partition size, nor is there a maximum: the only restriction on partitioning is that the partition widths and lengths must match along an axis of partitions. In Figure 5.8, we show an example of a valid partitioning layout, where the size of each partition represents the number of rows and columns contained within that partition. As the figure shows, the number of columns in each column of block partitions is the same width, and the number of rows in each row of block partitions is the same height. This example shows that Modin has flexible partitioning, none of the individual partition shapes need to match.

Virtual Partitioning

Partitioning over blocks not only allows us to scale in both directions, but it also enables support of traditionally difficult or expensive operations, like `transpose`. However, partitioning in blocks separates entire rows and columns from each other, and many operators in the Modin dataframe algebra are defined such that they require access to an entire row or column. For example, the user-defined function that is passed to the `map` operator assumes it has full row or column access, depending on the `axis` parameter. To enable all types of access patterns, we implemented **virtual partitioning**.

With virtual partitioning, Modin developers are able to write algorithms and optimizations in whatever partitioning format is most natural, no matter the current physical partitioning. Our motivation for providing this degree of flexibility was to enable faster development and implementation of new algorithms, and to lower the barrier to implementing new algorithms; developers do not even need to understand the partitioning mechanisms or handle low level details of the partitioning mechanism themselves. Since even distributed systems developers do not like having to deal with partitioning, we allow them access to row partitioning and column partitioning via simple internal APIs.

Partition Placement and Shuffling

The internal partitioning mechanisms in the partitioning layer are also responsible for their physical placement, to include shuffling data between worker nodes. Here we discuss at a high level the policies in place for data movement. Partitions are first placed in the same node as those partitions within the same column. The placement policy is programmable, but we have found that column accesses and manipulations in pandas are far more common than row manipulations. When the developer requests to perform an operation along the rows and requires that data be moved between the nodes, the shuffling mechanism will ensure that all of the data for each row partition is moved to the same node.

Given the flexibility of the partitioning mechanism and the need to potentially shuffle data between every operator in the worst case, there is a significant amount of future work that will be needed to enable ideal partitioning and data shuffling for dataframes.

5.7 Execution and Scheduling

The Execution Layer is responsible for serializing and executing code on one or more input dataframe partitions. This layer is responsible because there is a significant amount of overlap between task-parallel execution engines and schedulers. For the reference implementation in Modin, we have enabled support for both Ray and Dask.

In the effort of enabling data scientists to be productive, this layer is also important because data scientists may not have a choice of what execution engine or cluster resources they have access to. The purpose of Modin is to allow users to run the same notebooks on the hardware and systems that are available to them. We now discuss details about Modin's execution layer for both Ray and Dask.

Execution on Ray

Ray is an execution engine that exposes two main abstractions: tasks and actors. The actor abstraction in Ray allows for stateful computation, and while this may seem like a natural fit for data-intensive applications, we have found that in practice that there are significant drawbacks to using Ray's actor abstraction to implement a distributed dataframe. First, each actor must have a handle, or line of direct communication, to every other actor for high performing shuffles. When a new compute node becomes available or goes down, as is common in elastic clusters or serverless environments, bringing that node up and allowing it to accept new computation becomes more expensive than anonymous tasks. Additionally, at the time of evaluation, we found that an implementation on Ray's actor startup time to cause performance be roughly 30% slower than anonymous tasks. Ray has a distributed scheduler, allowing it to scale well and prevent bottlenecks in the driver process.

Modin's Ray engine currently uses Ray's object store, but there are efforts planned to abstract the object store away from the Ray engine implementation in Modin so that we can also make use of high performance object stores. Ray's object store is immutable, which allows us to treat each task's output independently and aligns with Modin's overall architecture.

Ray has a unique Python decorator-style API for remote task submission, so naively enabling compute kernels to execute against a Ray engine would force us to define functions independently for Ray. Independently defining compute kernels specific for Ray would not allow us to reuse code between execution engines, and is altogether extremely code-intensive and difficult to maintain. Instead, we have one internal remote task declaration that has the required decorator and we serialize the compute kernel defined in the query compiler to ship it to the task. When the task receives it, the function gets deserialized and applied to a partition of specific data. This detail is important to understanding how Modin's abstractions allow for runtime-specific details to be implemented without directly affecting other layers or runtimes.

Execution on Dask

Dask, like Ray, exposes task and actor abstractions. Dask's API accepts the compiled compute kernel directly. Dask has a single, global scheduler with no fault tolerance, which is both a performance bottleneck for scheduling and a single point of failure, but gives . Given its position in the Python community, many data scientists are either familiar with Dask or are using it, so it is naturally a backend that would be useful to support in Modin.

Dask stores task outputs as futures within the worker's Python process memory. In practice, this means that the objects that are stored as task outputs by Dask are mutable, and any side effects created by compiled compute kernels can mutate partition data. This also means that tasks in Dask are not truly anonymous since any task may modify its input inplace. This is a problem because users can themselves define functions that modify dataframes in place via the pandas API `apply`, which is an essentially unbounded user defined function. In order to avoid the consequences of destructive UDFs in Dask, we must first make a copy of the data before a user's function can be applied.

Execution on Omnisci

Modin also have enabled a purely relational type storage through the inclusion of OmniSciDB as a blackbox. It is not controlled directly through the storage layer API but indirectly through the execution. All the operations still go from the user facing API through the compiler API but the storage is completely controlled withing the OmniSci DB engine. Such an approach is not canonical for Modin but allows to offload the optimizations to the more performant engine and handle it more effectively in the case of GPU execution.

5.8 Related Work

Historically, many implementations have attempted to solve the problems of scaling dataframes, but are limited in different aspects: whether it be by not supporting all dataframe functions, or even by changing the underlying data model altogether. To the best of our knowledge, Modin is the first dataframe system that supports the dataframe's flexible data model and operations with correct and consistent semantics, while enabling dataframe operations to be parallelized at scale using formalized and flexible decomposition rules.

Systems that support dataframe operations. Original implementations of dataframe systems include pandas [81] and R [90]. Dataframes first originated in S [20], and was inherited by R. R dataframes suffer similar limitations to pandas in that they cannot exceed main-memory [117] and run on a single thread. Projects like Tidyverse [125] remove some of the properties of R dataframes to make them more like relational tables. R dataframe operators can be similarly made to run in-parallel via the decomposition rules we describe in Section 3.4. Dask Dataframe [30] partitions a dataframe along rows to make operations along the row axis more scalable, similar to a relational database. Vaex [119] is a system for imperatively querying static memory-mapped HDF5 files, supporting around 35-40% of the functionalities of the pandas API, similar to embedded databases like SQLite [50] and DuckDB [91].

There are many systems that support a subset of the pandas API via relational databases using various flavors of SQL. Koalas [63], an open-source project, translates 55% of the pandas into the API Spark SQL API via ANSII SQL. Ibis [53] translates a small subset of the pandas API into a variety of database backends, including ClickHouse [24], OmnisciDB [74], and more. Recently, there is work on choosing database backends [55] and translation into database systems like A-frame [107], Grizzly [59], and AIDA [27]. RIOT [131] achieved similar goals of employing a database backend for operating on R data beyond main-memory.

Here, we operationalize and extend the algebra in Chapter 3, and introduce formal decomposition rules and metadata management techniques. We also recently introduced the notion of *opportunistic evaluation* where we "batch" dataframe operations together to be executed in the background asynchronously, prioritizing what the user wants to see [127]; this is orthogonal to the techniques proposed in this paper—we expect both techniques to work together well.

Parallel/distributed database systems. Many parallel and distributed databases [80], such as Teradata [115], HadoopDB [4], and SparkSQL [9], partition data into rows using hash or range-based partitioning to parallelize row-oriented relational operators. Additionally, column stores, like C-Store [112], Dremel [72], MonetDB [16], BigTable [21], and HBase [122], partition the data along columns to better compress data and accelerate large-scale data analysis. Recent parallel relational and non-relational query processing systems include Google's BigQuery [71], Amazon's RedShift [44], Azure Synapse [6], Snowflake [28], Impala [15], MongoDB [22], among others. While these systems employ row/column-oriented partitioning to parallelize the query execution, they focus on unordered row-oriented operators and do not consider metadata operators. Modin needs to optimize operators that query and update metadata, and operate on the granularity of rows, columns, and blocks of cells. In addition, efficiently supporting mixed types is not covered by these systems.

Matrix computing and decomposition. Matrix partitioning and decomposition [132, 17, 79], commonly used to parallelize machine learning and scientific computing applications, is similar to dataframes in that it needs to support row-wise, column-wise, and cell-wise decomposition patterns. However, typical matrix decompositions are tailored for sparse matrices, and these systems generally don't support operators like joins, filters, group-bys, or heterogeneous data types. Array databases, like SciDB [111], target structured workloads, with well defined schemas optimized for scientific workloads, making them ill-suited for handling the flexible dataframe data model. There is additional work bridging relational and linear algebra, with a focus on supporting a small kernel of operators spanning both, in LaraDB [52], which we have drawn inspiration from in our past work.

5.9 Discussion

In this chapter, we targeted the dual challenges of scalability and semantics underlying dataframes. We introduced multiple flexible rule-based decomposition techniques for parallelizing dataframe operations across both row and column axes, and label, order, and type management techniques that help ensure metadata independence. Together, these techniques, as implemented in Modin, enable it to support pandas operations across both rows and columns at scale, while not compromising on pandas operation coverage, providing speedups of up to 50-100$\times$ relative to other partial and full dataframe implementations as we will see in Chapter 6.

Chapter 6

Evaluation

6.1 Introduction

In this chapter we evaluate Modin against other popular systems. We compare against Apache Spark/Koalas [9], Dask Dataframe [31], and pandas [81]. These systems were chosen based on their popularity and breadth of real-world use cases. The experiments conducted here focus on three system properties - first, we analyze the API parity with pandas in section 6.2, next we conduct micro benchmarks to compare the performance of individual dataframe methods in section 6.3, and lastly in section 6.4 we compare the scalability of each system.

6.2 Functional Evaluation

We first focus on the functional differences between these systems. To our knowledge, this is the first functional analysis of each systems parity with the pandas API. To determine whether a given pandas functionality is supported, we visited the documentation of each system, and in cases where this was inconclusive, we looked at the source code. We separate the analysis into 3 categories: I/O which focuses on supported data formats, metadata which focuses on support for metadata operations, and the dataframe API where we focus on overall coverage of the pandas dataframe API.

I/O

Data ingestion is paramount: it does not matter how fast a query can run if data cannot enter the system. Table 6.1 shows the implementation of operators provided by pandas. Several operators are not supported with native implementations in Modin, Dask or Koalas, we list

Table 6.1: I/O operator support in various systems. Operators are only considered if they have native implementations.

Operator	**pandas**	**Modin**	**Spark (Koalas)**	**Dask**
`read_csv`	×	×	×	×
`read_table`	×	×	×	×
`read_parquet`	×	×	×	×
`read_json`	×	×	×	×
`read_excel`	×	×		
`read_feather`	×	×		
`read_sql`	×	×	×	
`read_sql_table`	×	×	×	×
`read_sql_query`	×	×	×	
`read_fwf`	×	×		×
`read_hdf`	×	×		×
`read_orc`	×			×

Table 6.2: Metadata support in various systems. An asterisk indicates that a significant part of the operator is not possible in the system's architecture.

Operator	**pandas**	**Modin**	**Spark (Koalas)**	**Dask**
`mask` (by label)	×	×	×	×
`mask` (by position)	×	×	×	
`head`	×	×	×	×
`tail`	×	×	×	
User Orders	×	×		
`MultiIndex`	×	×		
`to_labels`	×	×	×	×
`from_labels`	×	×	×	

them here: `read_clipboard`, `read_sas`, `read_html`, `read_spss`, `read_gbq`, `read_pickle`, `read_stata`.

In this evaluation, we did not consider all possible parameters and extended support (e.g. reading files from S3) because these details are often not explicit. Therefore, any support for a format qualifies for Table 6.1.

Metadata

Metadata interaction is common in dataframe workloads. For example, selecting or filtering by label, finding the intersection of labels between two dataframes, or direct queries and

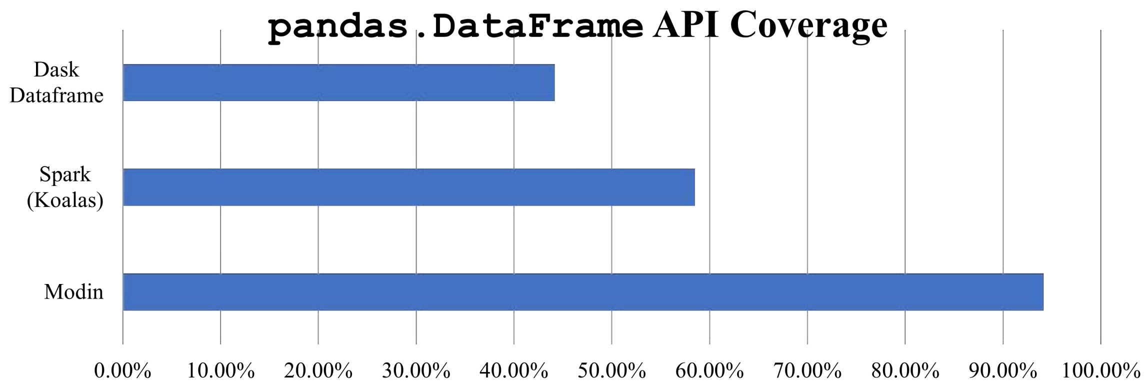


Figure 6.1: Percent coverage of the pandas dataframe API after deduplication.

manipulation on the labels. Table 6.2 shows the breakdown of the metadata behaviors and functionalities in dataframes. In each of these systems, `to_labels` is `set_index` and `from_labels` is `reset_index` [1].

The `mask` by label is equivalent to pandas `loc` operator, and similarly `mask` by position is equivalent to pandas `iloc` operator. Dask's implementation cannot support `iloc` because it does not support the user's preferred dataframe order.

Dataframe API

Figure 6.1 shows the relative API coverage for pandas, excluding identical behaviors. We exclude these to avoid skewing results toward implementing simple duplicated APIs. This also avoids double penalizing systems that are unable to implement duplicated APIs. Each argument is counted, such that a perfect score reflects that an operator is implemented for all valid argument combinations. For example, if an implementation is unable to implement one of the axes for any operator or class of operators, it will be given a 50% score for that operator provided it can implement the remaining arguments for the `axis` that is supported. Anywhere that there was ambiguity, the benefit of doubt was given to an implementation. The API details were gathered from the documentation pages of each system.

The difference in API coverage between Modin and the others can be mostly attributed to the architectures. Dask and Spark are both row stores, and so operators that rely heavily on communication between rows will be extremely inefficient or impossible. Koalas cannot

[1]In the case of Dask's implementation of `from_labels`, we found that it will not set the labels to the position notation, instead setting the labels within each partition to its local position notation. Since this violates the definition of `from_labels` and deviates from the original pandas implementation, we mark it as not implemented.

support anything outside of ANSII SQL because it is built on Spark SQL. Dask is not tied to SQL, but it is missing key metadata tracking and other features important for implementing several of the missing APIs.

Given the results in this section and how they contrast with the way the systems are marketed (“Pandas API on Spark” and “Pandas in the cloud” for Koalas and Dask, respectively.) we feel as though the data science and scientific communities would benefit from a more detailed review of these systems and their capabilities than what can be shown here.

6.3 Microbenchmark Performance Evaluation

In this section, we focus on individual operator performance of Modin against other systems.

Microbenchmarks against pandas

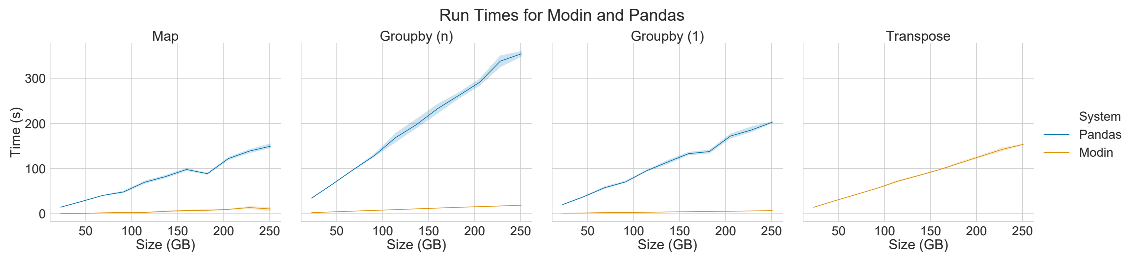


Figure 6.2: Each function shows runtime and 95% confidence region for both Modin and pandas. We omit pandas transpose as it is unable to scale beyond 6 GB.

To understand how the optimizations discussed in chapter 5 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 [77] 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.
- group-by (n): group by the non-null “passenger_count” column and count the number of rows in each group.
- group-by (1): count the number of non-null rows in the dataframe.

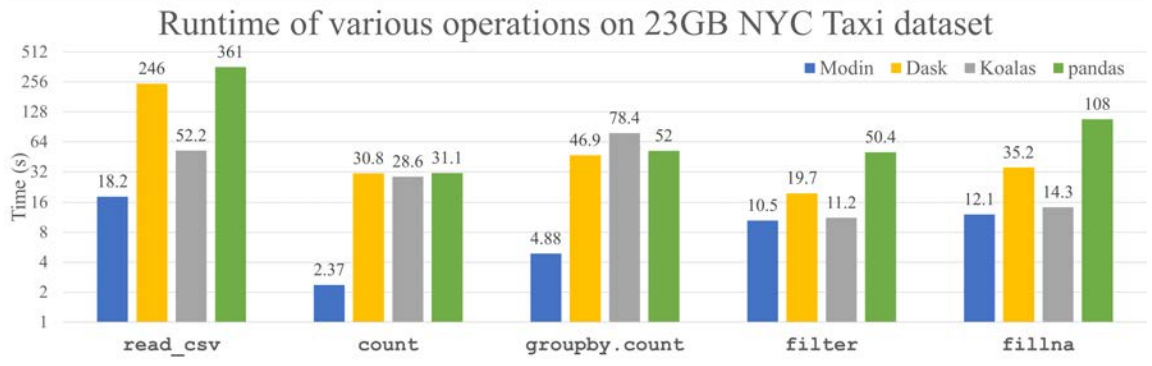


Figure 6.3: Single node performance of various operations on 23GB of data.

- 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 group-by(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 6.2 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, when performing 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 is about $12\times$ 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 ($\sim$150 million rows) after 2 hours. Through separate testing, we observed that pandas can only transpose dataframes of up to 6 GB ($\sim$6 million rows) on the hardware we used for testing.

Our preliminary case study and our experience with Modin demonstrates the promise of integrating simple optimizations to make dataframe systems scalable. Next, we compare microbenchmarks of Modin against other distributed systems with pandas as a baseline.

Microbenchmarks against other distributed systems

We now present microbenchmark performance of Dask, Koalas, and Modin. For Modin, we separate the performance plots into "Ray" and "Dask" engines. The Dask engine for Modin is not to be confused with the Dask dataframe, against which are comparing. Modin is able to use the Dask distributed scheduler, but does not share any code with Dask dataframe. We show two plots for performance: one for single node performance on a variety of operators, and another for multinode performance on the same operators. Aside from the expensive apply, the operators here were chosen because they are implemented by all systems.

Figure 6.3 shows the results of the microbenchmark performance. These experiments were run on a x1e.32xlarge with 128 CPUs and 4TB of memory. It is important to note here that Koalas was not able to run with default settings[2], considerable effort was made to enable and optimize Koalas in this environment. Dask was run with the recommended settings from the documentation, and many different configurations were attempted. Modin was run with default settings, and there is still room here for further optimization in the compute kernels themselves as well as some tuning.

The major differences in the performance come down to architectural decisions made in the frameworks. Dask and Spark (Koalas) both have chosen a rigid, strict row partitioning architecture that makes it difficult to handle things like data skew, common in groupby operations. There is also the issue of order: Spark will often reorder rows during shuffling, so in order to match the user's expectation of order, an extra sort would be required. It is important to mention here that the only sort available in Dask dataframe happens implicitly when an index is created from a column, which diverges from user expectations and the dataframe data model. Dask does not allow any order of the data other than the lexicographical order of the row labels.

Modin offers more functionality than is possible in Dask and Koalas, and has better performance. There is still significant room for improved performance in Modin from faster compute kernels to more efficient runtimes and an optimizer. If Spark and Dask dataframe were modified to have better support for user orders and more flexible partitioning, they would be able to benefit from the existing optimizations in Modin. We have already implemented a Dask engine for Modin, however it performs roughly 10-20% slower than the Ray engine due to the issues described in Section 5.7.

6.4 Scaling Performance against baselines

Our experiments in this section address the following questions:

- Compared to existing dataframe systems, including Koalas [63], Dask Dataframe [30], and pandas [81], how well does Modin scale dataframe operations over a large number of CPU cores?

- How much do the optimization techniques, eager data pipelining and selective data exchange, reduce the execution time? (Section 5.4)

[2]Koalas consistently ran out of memory or forced all of the data onto a single partition.

All experiments are run on an AWS instance x1e.32xlarge that has 3904 GB of main memory and 128 vCores. The OS of the test machine is configured to be Ubuntu 20.04.

Experiment setup

Benchmark. We use the NYC Yellow Taxi Dataset 2015 [77] in our experiments. It includes the history of taxi trips in the city. This dataset has 150 million rows and 20 columns, occupying 23GB on disk. We use this dataset to test the scalability of several widely-used dataframe functions, including `read_csv`, `fillna`, `count`, `groupby` followed by `count`, `join`, and `median`. We choose these functions because they cover many stages and aspects of a typical data science lifecycle, such as data ingestion (e.g., `read_csv`), data cleaning (e.g., `fillna`), and data analysis (e.g., `join`). We additionally test two Modin operators that query and manipulate metadata: `from_labels` and `infer_types`. We also use this dataset to test the optimization opportunities when choosing the best rewriting rules, as discussed in Section 5.4.

Baselines. We compare Modin with three popular dataframe systems, pandas [81], Koalas [63], and Dask Dataframe [30] (denoted as `Dask DF` in this section)—not to be confused with the Dask parallel compute framework [31].

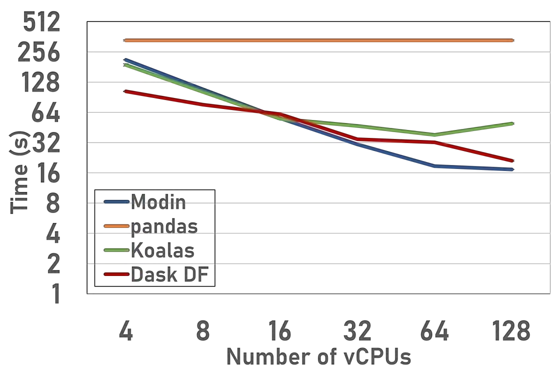
Scalability of operators supported by Modin and the baselines

We first test the scalability of Dask DF, Koalas, pandas, and Modin for operators supported by all systems, including: `read_csv`, `fillna` replacing the `NULL` values for each row, `count` counting the non-`NULL` values for all columns, and `groupby.count` using the "passenger_count" column as the group key. We vary the number of vCPUs used by each system and report the execution time.

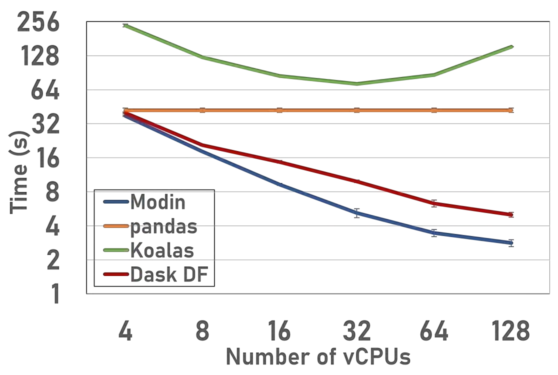
It is important to note here that Koalas was not able to run with default settings[3], and considerable effort was made to enable and optimize Koalas in this environment. We also tuned the Dask DF configurations; the results from the best performing Dask DF configuration are reported. The difference between the default performance and the best case performance in Dask DF was between 10× and 40×. Modin is run with default settings.

Figure 6.4 shows the test results. We see that Modin has the lowest execution time compared to the baselines for all operators. This is mainly because Modin parallelizes these operators and lazily computes the metadata (e.g., computing the type information). pandas does not scale because it runs on a single thread. Koalas and Dask DF can scale

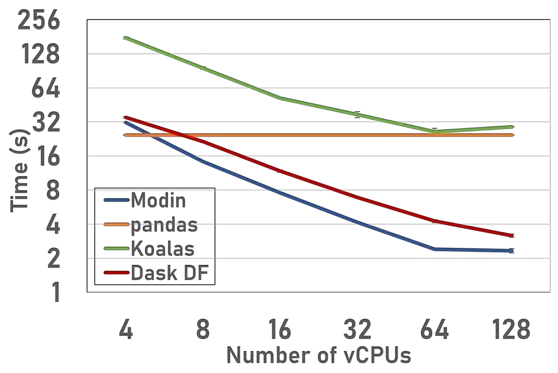
[3]Koalas consistently ran out of memory or forced all of the data onto a single partition on default settings, so many attempts at optimization were made to ensure a fair comparison.



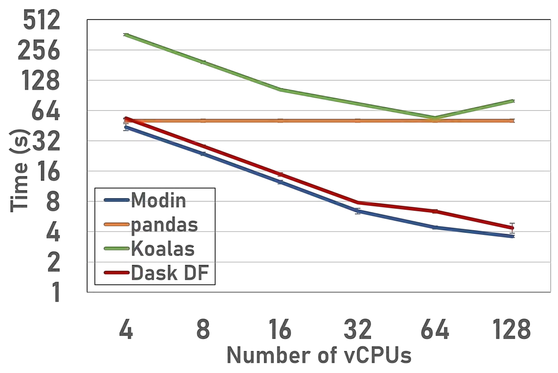
(a) `read_csv`



(b) `fillna` (`map` along rows)



(c) `count` (`reduce`)



(d) `groupby.count` (`groupby`)

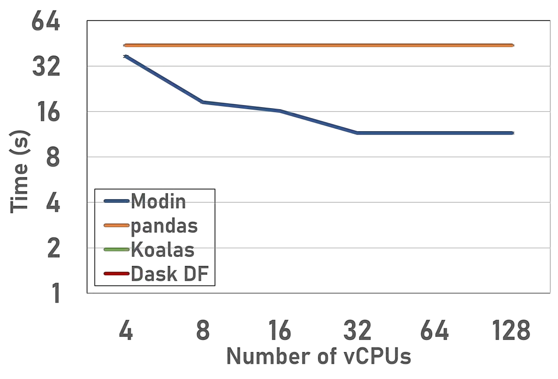
Figure 6.4: Scale test for operations that are supported by all tested systems.

these operators because these operators can be implemented using row-wise decomposition. Koalas has higher execution time than pandas and other systems for operators `fillna`, `count`, and `groupby.count` mainly due to the overhead of Spark and an extra phase of sorting the output rows to maintain the natural order. For example, Koalas parallelizes `fillna` by applying `fillna` function for each partition of rows. Since Spark does not maintain the ordering information across partitions, Koalas needs to sort the output rows as the order of the input rows.

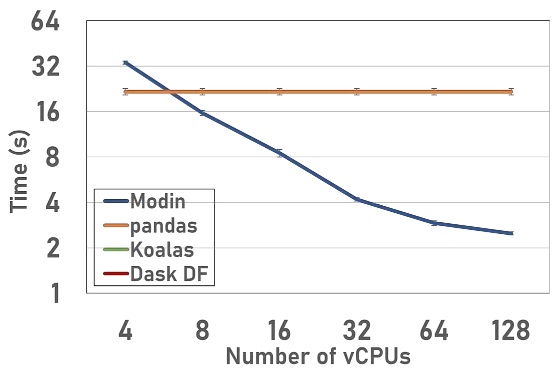
Scalability of operators not supported by all baselines

We now test operators that are not supported by all baselines, including `median`, `from_labels`, `infer_types`, and `join`. The baselines do not support these operators because they do not support operating on the column axis (e.g., computing the median for each column), the systems run out of memory (e.g., `join` for Dask DF), and

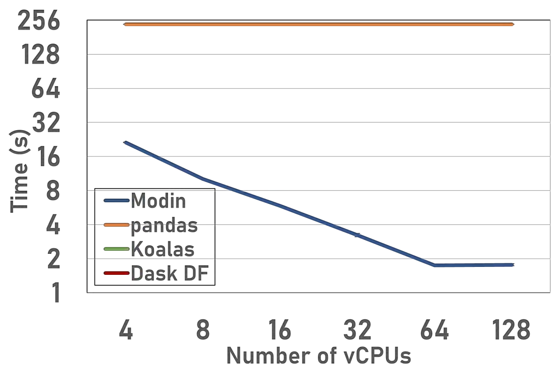
Figure 6.5: Performance of operations only supported in Modin and pandas.



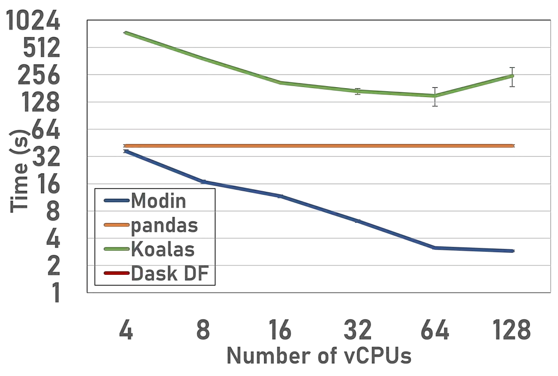
(a) `median`



(b) `from_labels`



(c) `infer_types`



(d) `join`

they do not support querying and manipulating metadata (e.g., `from_labels`). We vary the number of vCPUs and report the execution time of each operator.

Since Dask DF and Koalas are row-store-based dataframe systems, they do not support computing median for each column. Figure 6.5a shows the scalability of computing the `median` for every numeric column. The time reported includes a filter on the types of the columns to select only numeric columns. In this case, the parallelism Modin can exploit is limited by the number of column in the dataset, so increasing the number of cores beyond 20 (the number of columns in the dataset) does not improve the performance.

Figure 6.5b and Figure 6.5c show the results of `from_labels` and `infer_types`, respectively. `infer_types` is configured to infer the types of all columns. Dask DF and Koalas do not support the two metadata operators. `from_labels` in Modin has the overhead of inferring the positions of the labels and inserting them as data compared to

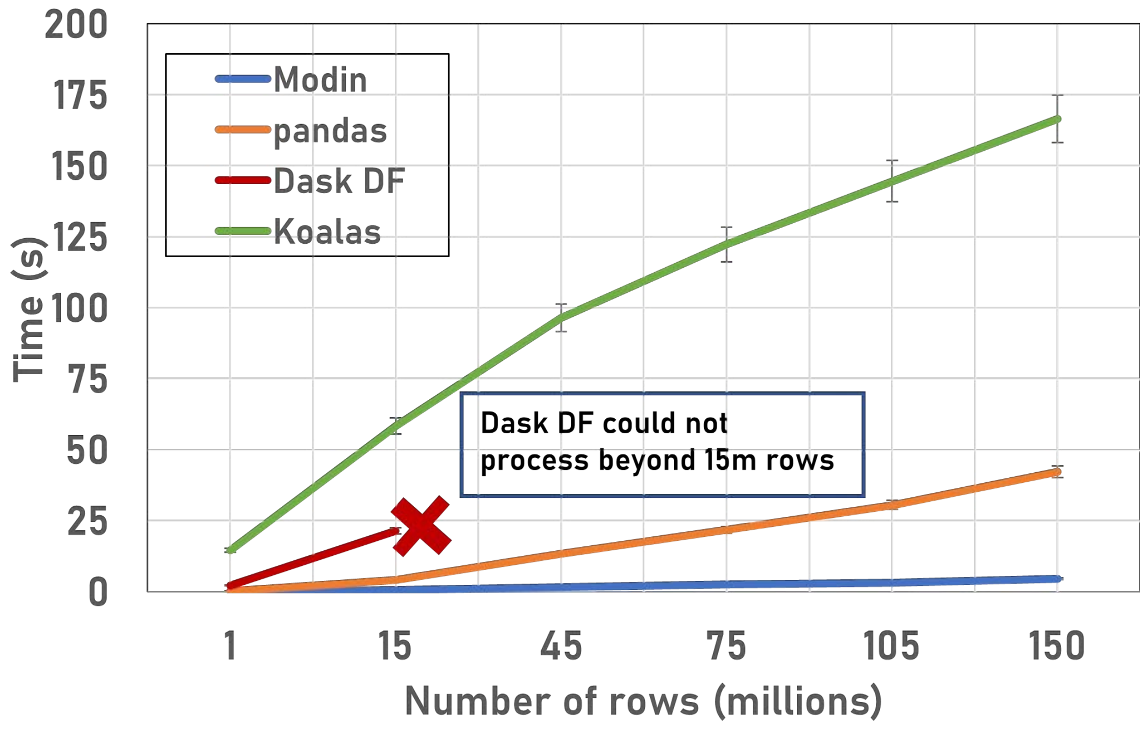


Figure 6.6: Join performance under varied number of rows

pandas, which eagerly materializes the positions. Therefore, at a smaller number of cores, the overhead of inferring the positions dominates and Modin has higher execution time than pandas. However, as the number of cores increases, this overhead can be amortized. Modin can scale this operator and achieve up to a 10x faster runtime than pandas. Modin prevails over pandas for the `infer_types` operator because it decomposes and parallelizes the execution of `infer_types` and uses indexes in our type system to quickly determine the type information. We see the performance improvement of Modin over pandas is up to $100\times$.

We also tested a self-join on the row labels of the NYC dataset. Dask DF runs out of memory for the `join` operator, so it is not included in the results shown in Figure 6.5d. We see that Modin has lower execution time than both pandas and Koalas. While Koalas can reduce the execution time as the number of cores increases, the overhead of Spark dominates and Koalas is slower than pandas for `join`.

To compare the `join` performance of Dask DF with Modin and other systems, we perform another experiment that varies the number of rows of the test dataset. Figure 6.6

shows the results. We see that Dask DF runs out of memory when we use more than 15 million rows. For the case of 15 million rows, Modin performs 50$\times$ faster than Dask DF.

6.5 Discussion

In this chapter, we evaluated Modin against several established systems: pandas, Dask Dataframes, and Koalas. Modin was shown to cover more functionalities than the other systems due to the theoretical foundation and architecture presented in Chapters 3 and 4. We also showed that Modin is able to perform dataframe queries at faster speeds than established systems due to metadata independence and improved parallelization.

The next chapter explores the potential optimizations unlocked with the architecture presented in Section 4, with the added benefit of leveraging think time to perform background computation.

Chapter 7

Architecture Case Study: Opportunistic Evaluation

7.1 Introduction

During the course of Machine Learning (ML) model development, a critical first step is *data validation*, ensuring that the data meets acceptable standards necessary for input into ML training procedures. Data validation involves various sub-tasks, including *data preparation*: transforming the data into a structured form suitable for the desired end-goal, and *data cleaning:* inspecting and fixing potential sources of errors. These validation steps of data preparation and cleaning are essential even if the eventual goal is simply exploratory data analysis as opposed to ML model development—in both cases, the quality of the eventual end-result, be it models or insights, are highly tied to these steps. This data validation process is highly exploratory and iterative, as the data scientist often starts off with a limited understanding of the data content and quality. Data scientists therefore perform data validation through *incremental trial-and-error*, with the goals evolving over time: they make a change, inspect the result (often just a sample) to see if it has improved or "enriched" the dataset in some way, e.g., by removing outliers or filling in NULL values, expanding out a nested representation to a flat relational one, or pivoting to organize the dataset in a different manner more aligned with the analysis goals.

To support this iterative process of trial-and-error, data scientists often use powerful data analysis libraries such as pandas [81] within computational notebooks, such as Jupyter or Google Colab [61, 41]. Pandas supports a rich set of incrementally specified operators atop a tolerant dataframe-based data model, drawn from relational algebra, linear algebra, and spreadsheets [87] embedded within a traditional imperative programming language, Python. While the use of dataframe libraries on computational notebooks is a powerful solution for data validation on small datasets, this approach starts to break down on larger datasets [87], with many operations requiring users to wait for unacceptably long periods, breaking flow. Currently, this challenge may be overcome by either switching to a *distributed dataframe system* (such as Dask [32] and Modin [73]), which introduces setup overhead and potential incompatibilities with the user's current workflow, or by users manually optimizing their queries, which is a daunting task as pandas has over 200 dataframe operations. We identify two key opportunities for improving the interactive user experience *without requiring changes to user behavior*:

- Users often do not want to inspect the entire results of every single step.

- Users spend time thinking about what action to perform next.

Unfortunately, at present, every cell (the unit of execution in a notebook) issued by the user is executed verbatim immediately, with the user waiting until execution is complete to begin their next step. Moreover, the system is idle during *think time*, i.e., when users are thinking about their next step or writing code. Fundamentally, *specification* (how the user writes the query) and *execution* (what the system executes) are tightly coupled.

In this paper, we outline our initial insights and results towards optimizing dataframe queries for interactive workloads by *decoupling specification and execution*. In particular, dataframe queries are not executed immediately, unless the user intends to inspect the results, but are deferred to be computed during think time. We distinguish operators that produce results that users inspect, what we call *interactions*, from those that do not. We can then use program slicing to quickly determine what code is *critical* in that it influences the interactions, i.e., what the user intends to see immediately, and what is *non-critical*, in that it can be computed in the background during think-time to speed up future interactions. For the critical portions, we further identify if it can be rewritten in ways that allows us to improve interactivity further. For example, identifying that users often only examine the first or last few rows/columns of the result allows us to compute this as part of the critical portion and defer the rest to the non-critical portion. For the non-critical portions, by deferring the execution of the non-critical portions, we can perform more holistic query planning and optimization. Moreover, we may also *speculatively* compute other results that may prove useful in subsequent processing. We call our framework *opportunistic evaluation*, first discussed in Chapter 2. Opportunistic evaluation preserves the benefits of eager evaluation (in that critical portions are prioritized), and lazy or deferred evaluation (in that non-critical portions are deferred for later computation). This paper builds on Chapters 3 and 4, wherein we outline our first steps towards establishing a formal framework for reasoning about dataframe optimization systematically.

7.2 Background and Motivation

Key Concepts

Users author dataframe queries in Jupyter notebooks, comprising code cells and output from executing these code cells. Figure 7.1a shows an example notebook containing dataframe queries. Each code cell contains one or more queries and sometimes ends with a query that outputs results. In Figure 7.1a, every cell ends in a query (namely, `df1.describe()`, `df1.head()`, and `df2.describe()`) that outputs results. Dataframe queries are comprised of operators such as `apply` (applying a user defined function on rows/columns), `describe` (compute and show summary statistics), and `head` (retrieve the top K rows of the dataframe). Operators such as `head` and `describe`, or simply the dataframe variable itself, are used for inspecting intermediate results. We call these operators ***interactions***. Users construct queries incrementally by introducing interactions to verify intermediate results. An interaction usually depends on only a subset of the operators specified before it. For example, `df1.describe()` in Figure 7.1a depends only on `df1 = pd.read_csv("small_file")` but not `df2 = pd.read_csv("LARGE_FILE")`. We call the set of dependencies of an interaction the ***interaction critical path***. To show the results of a particular interaction, the operators not on its interaction critical path do not need to be executed even if they were specified before the interaction.

After an interaction, users spend time inspecting the output and authoring new queries based on the output. We call the time between the display of the output and the submission of the next query ***think time***, during which the CPU is idle (assuming there are no other processes running on the same server) while the user inspects intermediate results and authors new queries. We propose ***opportunistic evaluation***, an optimization framework that leverages this think timeto reduce interactive latency. In this framework, the execution of operators that are not on interaction critical paths, which we call *non-critical operators*, are deferred to being evaluated asynchronously during think timeto speed up future interactions.

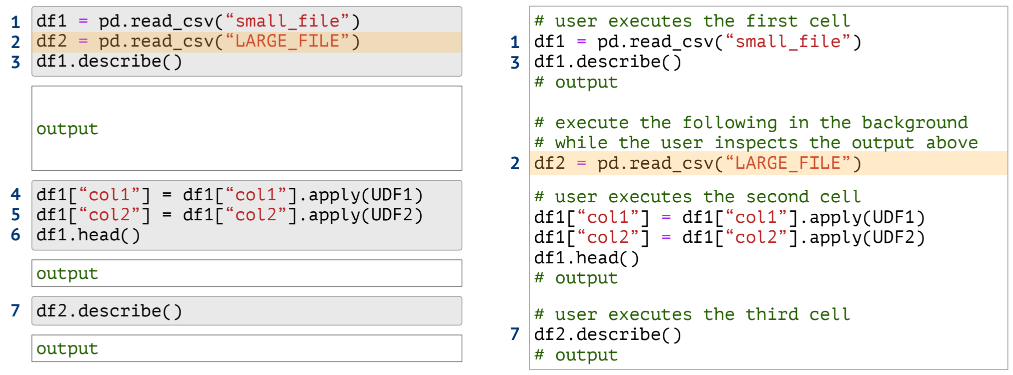
Motivating Scenarios and Example Optimizations

To better illustrate the optimization potential of opportunistic evaluation, we present two typical data analysis scenarios that could benefit from asynchronous execution of queries during think timeto minimize interactive latency. While the user's program remains the same, we illustrate the modifications to the execution plan that highlights the transformations made.

Interaction-based Reordering

Consider a common workflow of analyzing multiple data files, shown in Figure 7.1a. The user, Sam, executes the first cell, which loads both of the files, and is forced to wait for *both* to finish loading before she can interact with *either* of the dataframes. To reduce the interactive latency (as perceived by the user), we could conceptually re-order the code to optimize for the immediate output. As shown in Figure 7.1b, the re-ordered program defers loading the large file to after the interaction, `df1.describe()`, obviating the need to wait for the large file to load into `df2` before Sam can start inspecting the content of the small file.

To further reduce the interactive latency, the system could load `df2` while Sam is viewing the results of `df1.describe()`. This way, the time-consuming process of loading the large file is completed during Sam's think time, thus reducing the latency for interacting with `df2`.



```
1 df1 = pd.read_csv("small_file")
2 df2 = pd.read_csv("LARGE_FILE")
3 df1.describe()

  output

4 df1["col1"] = df1["col1"].apply(UDF1)
5 df1["col2"] = df1["col2"].apply(UDF2)
6 df1.head()

  output

7 df2.describe()

  output
```

(a) Original program where user has to wait for both files to load before viewing any.

```
  # user executes the first cell
1 df1 = pd.read_csv("small_file")
3 df1.describe()
  # output

  # execute the following in the background
  # while the user inspects the output above
2 df2 = pd.read_csv("LARGE_FILE")

  # user executes the second cell
  df1["col1"] = df1["col1"].apply(UDF1)
  df1["col2"] = df1["col2"].apply(UDF2)
  df1.head()
  # output

  # user executes the third cell
7 df2.describe()
  # output
```

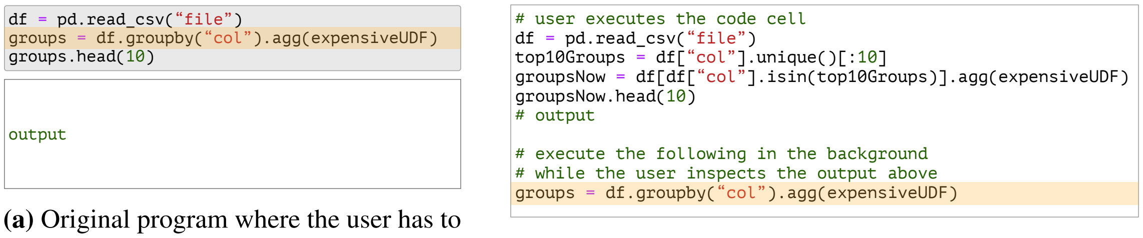
(b) Optimized program where the user can view the smaller file first while the other loads.

Figure 7.1: Example program transformation involving operator reordering.

Prioritizing Partial Results

For any large dataframes, users can only inspect a handful of rows at a time. However the current evaluation mechanism requires *all* the rows to be evaluated. Expensive queries such as those involving user-defined functions (UDFs) could take a long time to fully compute, as shown in Figure 7.2a.

To reduce interactive latency, one can prioritize computation of only the portion of the dataframe inspected. This method is essentially an application of *predicate pushdown*, a standard technique from database query optimization. Figure 7.2b provides an example transformation for the particular operator, `groupby`. While the first cell prioritizes the computation of the inspected rows, the user may still need the result of the entire computation, which is scheduled to be computed later while Sam is still reading the result of the previous cell, `groupNow.head(10)`, i.e. the think time. A noteworthy attribute of dataframes is row and column equivalence, introduced in Chapter 3, which means that predicate pushdown can also happen when projecting columns as well.



```
df = pd.read_csv("file")
groups = df.groupby("col").agg(expensiveUDF)
groups.head(10)
```

```
output
```

(a) Original program where the user has to wait for an expensive UDFs to fully compute.

```
# user executes the code cell
df = pd.read_csv("file")
top10Groups = df["col"].unique()[:10]
groupsNow = df[df["col"].isin(top10Groups)].agg(expensiveUDF)
groupsNow.head(10)
# output

# execute the following in the background
# while the user inspects the output above
groups = df.groupby("col").agg(expensiveUDF)
```

(b) Optimized program where the user can view a partial result sooner.

Figure 7.2: Program transformation involving predicate pushdown.

7.3 Assessment of Opportunities with Notebook Execution Traces

To assess the size of opportunity for our aforementioned optimizations to reduce interactive latency in computational notebooks, we evaluate two real world notebook corpora.

One corpus is collected from students in the **Data 100** class offered at UC Berkeley. Data 100 is an intermediate data science course offered at the undergraduate level, covering topics on tools and methods for data analysis and machine learning. This corpus contains 210 notebooks across four different assignments, complete with the *history* of cell execution content and completion times captured by instrumenting a custom Jupyter extension.

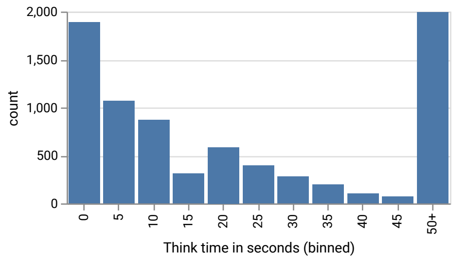
We also collected Jupyter notebooks from **Github** comprising a more diverse group of users than Data 100. Jupyter's IPython kernel stores the code corresponding to each individual cell executions in a local `history.sqlite` file[1]. We used 429 notebook execution histories that Macke et al. [67] scraped from Github that also contained pandas operations.

To assess optimization opportunities, we first quantify think timebetween cell executions, and then evaluate the prevalence of the code patterns discussed in Section 7.2.

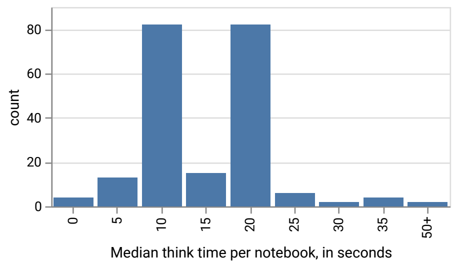
[1]`https://ipython.readthedocs.io/en/stable/api/generated/IPython.core.history.html`

Think-Time Opportunities

Our proposed opportunistic evaluation framework takes advantage of user think timeto asynchronously process non-critical operators to reduce the latency of future interactions. To quantify think time, we measure the time lapsed between the completion of a cell execution and the start of the next cell execution using the timestamps in the cell execution and completion records, as collected by our Jupyter notebook extension. Note that the think timestatistics are collected only on the Data 100 corpus, as the timestamp information is not available in the Github corpus. Figure 7.3a shows the distribution of think timeintervals, measured in seconds, between consecutive cell executions across all notebooks, while Figure 7.3b shows the distribution of the median think timeintervals, measured in seconds, within each notebook. We removed automatic cell re-execution ("run all") from the dataset. We can see that while there are many cells that were executed quickly, there exist cells that had ample think time—the 75th percentile think timeis 23 seconds.



(a) Think timebetween cell executions.

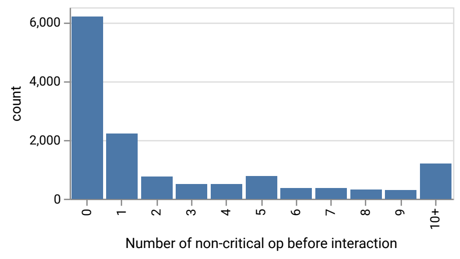


(b) Median think timeper notebook across cells.

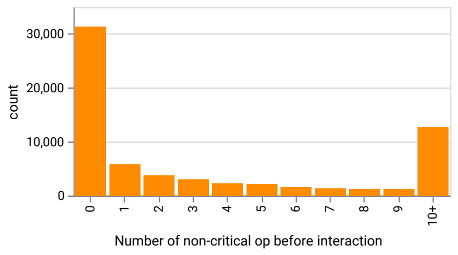
Figure 7.3: Think timethe average "think time" between cell executions and the average think timeper notebook.

Program Transformation Opportunities

Interaction-Based Reordering. To assess the opportunities to apply operator reordering to prioritize interactions, we evaluate the number of non-critical operators specified before each interaction. We use the operator DAG, to be described in Section 7.4, to determine the dependencies of an interaction and count the number of operators that are not dependencies, i.e., non-critical operators, specified above the interaction. Figure 7.4 shows the distributions for the two datasets. In both cases, non-critical operators present a major opportunity: the Data 100 and Github corpus have, respectively, 54% and 42% interactions with non-critical operators.

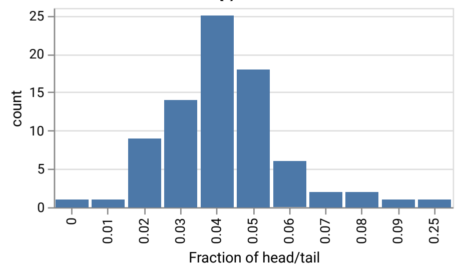


(**a**) Data 100: $\mu = 4$, $\sigma = 5$

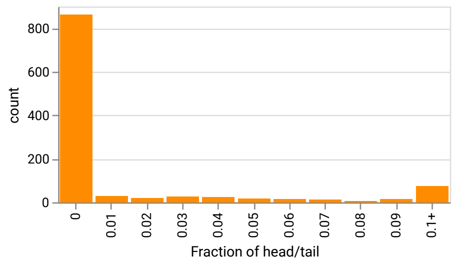


(**b**) Github: $\mu = 7$, $\sigma = 11$

Figure 7.4: Number of non-critical operators before interactions.

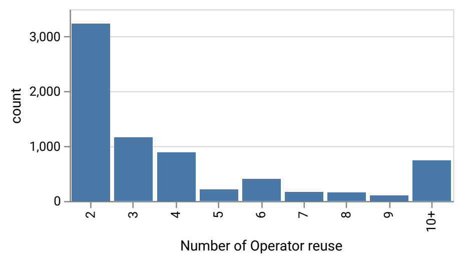


(**a**) Data 100: $\mu = 0.04$, $\sigma = 0.028$

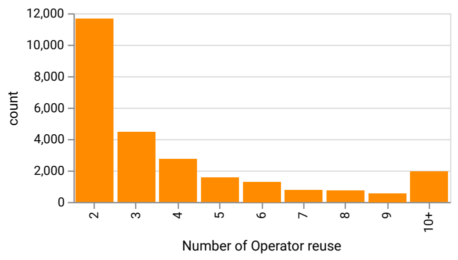


(**b**) Github: $\mu = 0.11$, $\sigma = 0.21$

Figure 7.5: Stats for head/tail interactions used in each notebook.



(**a**) Data 100: $\mu = 5$, $\eta = 3$, $\sigma = 8$



(**b**) Github: $\mu = 7$, $\eta = 3$, $\sigma = 14$

Figure 7.6: Distribution of number of operators that can benefit from reuse.

Prioritizing Partial Results The optimization for prioritizing partial results via predicate pushdown can be applied effectively to many cases when predicates are involved in queries with multiple operators. The most common predicates in the dataframe setting are `head()` and `tail()`, which show the top and bottom K rows of the dataframe, respectively. Figure 7.5b and Figure 7.5a show the distribution of the fraction of interactions that are either `head` or `tail` in each notebook. We see that partial results views are much more common in the GitHub dataset than Data 100. This could be due to the fact that users on GitHub tend to keep the cell output area short for better rendering of the notebook by Github, but further studies are needed to corroborate this hypothesis. Lastly, partial views are not nearly as prevalent as non-critical operators before an interaction, accounting only for $< 20\%$ of the interactions.

Reuse of Intermediate Results Since dataframe queries are incrementally constructed, with subsequent queries building on top of previous ones, another common query optimization technique that is applicable is caching these intermediate results. To assess the opportunities to speed up queries by caching, we evaluate the number of times an operator is shared by different interactions but not stored as a variable by the user. Ideally, we would also have the execution times of the individual operators, which is not possible without a full replay. We present an initial analysis that only assesses the *existence* of reuse opportunities, as shown in Figure 7.6b and Figure 7.6a. Both the Data 100 and Github datasets have a median of 3 operators that can benefit from reuse.

Of the types of optimizations explored, operator reordering appears to be the most common. Thus, we focus our initial explorations of opportunistic evaluation on operator reordering for asynchronous execution during think time, while supporting preemption to interrupt asynchronous execution and prioritize interaction.

7.4 System Architecture

In this section, we introduce the system architecture for implementing our opportunistic evaluation framework for dataframe query optimization within Jupyter notebooks. At a high level, we create a custom Jupyter Kernel to intercept dataframe queries in order to defer, schedule, and optimize them transparently. The query execution engine uses an operator DAG representation for scheduling and optimizing queries and caching results, and is responsible for scheduling asynchronous query executions during think time. When new interactions arrive, the execution of non-critical operators is preempted and partial results are cached to resume execution during the next think time. A garbage collector periodically uncaches results corresponding to the DAG nodes to avoid memory bloat based on the likelihood of reuse.

Kernel Instrumentation

Figure 7.7 illustrates the round-trip communication between the Jupyter front-end and the Python interactive shell. The black arrows indicate how communication is routed normally in Jupyter, whereas the green and purple arrows indicate how we augment the Jupyter

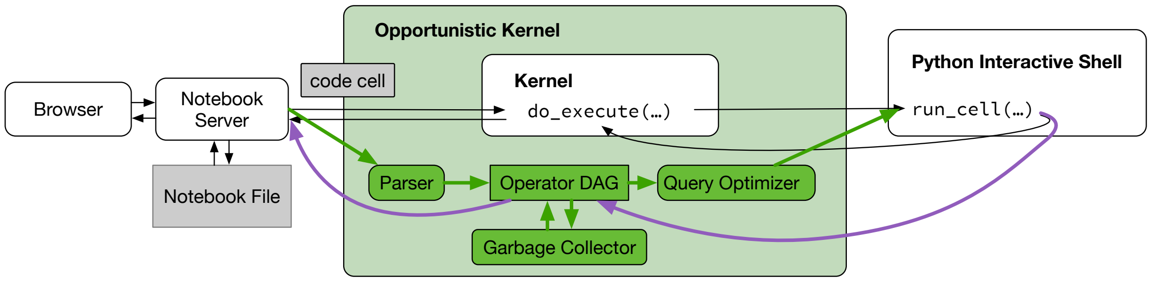


Figure 7.7: Opportunistic Evaluation Kernel Architecture.

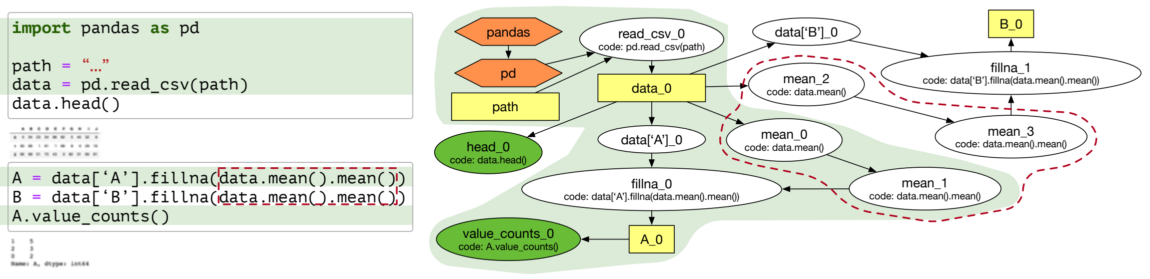
Kernel to enable opportunistic evaluation. First, when the code is passed from the front-end to the kernel, it is intercepted by the custom kernel that we created by wrapping the standard Jupyter kernel. As shown in the green box, the code is passed to a parser that generates a custom intermediate representation, the operator DAG. The operator DAG is then passed to the query optimizer to create a physical plan for the query to be executed. This plan in then passed to the Python interactive shell for execution. When the shell returns the result after execution, the result is intercepted by the custom kernel to augment the operator DAG with runtime statistics as well as partial results to be used by future queries, and the query results are passed back to the notebook server, as indicated by the purple arrows.

Intermediate Representation: Operator DAG

Figure 7.8 shows an example operator DAG constructed from the code snippet on the left. The orange hexagons are imports, yellow boxes are variables, ovals are operators, where green ovals are interactions. The operator DAG is automatically constructed by analyzing the abstract syntax tree of the code, in the parser component in Figure 7.7. We adopt the static single assignment form in our operator DAG node naming convention to avoid ambiguity of operator references, as the same operator can be invoked many times, either on the same or different dataframes. In the case that the operator DAG contains non-dataframe operators, we can simply project out the irrelevant operators by keeping only the nodes that are weakly connected to the `pandas` import node.

To see how the operator DAG can be used for optimization, consider two simple use cases:

Critical path identification. To identify the critical path to the interaction `A.value_counts()`, we can simply start at the corresponding node and traverse the DAG backwards to find all dependencies. Following this procedure, we would collect all nodes in the green region as the critical path to `A.value_counts()` (corresponding statements are highlighted in green on the left), slicing out the operators associated with



```
import pandas as pd

path = "..."
data = pd.read_csv(path)
data.head()
```

```
A = data['A'].fillna(data.mean().mean())
B = data['B'].fillna(data.mean().mean())
A.value_counts()
```

```
1    5
2    3
0    2
Name: A, dtype: int64
```

Figure 7.8: Example Code Snippet and Operator DAG.

the statement `B = data['B'].fillna(data.mean().mean())`, which does not need to be computed for the interaction.

Identifying repeated computation. Note that `data.mean().mean()` is a common subexpression in both `A` and `B`; recognizing this allows us to cache and reuse the result for `data.mean().mean()`, which is expensive since it requires visiting every element in the dataframe. We assume that operators are *idempotent*, i.e., calling the same operators on the same inputs would always produce the same results. Thus, descendants with identical code would contain the same results. Based on this assumption, we eliminate common subexpressions by starting at the root nodes and traversing the graph breadth first, merging any descendants with identical code. We then proceed to the descendants of the descendants and carry out the same procedure until the leaf nodes are reached. Following this procedure, we would merge `mean_0` with `mean_2` and `mean_1` with `mean_3` in the red dotted region in Figure 7.8.

Operator Execution & Garbage Collector

When a notebook cell is executed, the opportunistic kernel first parses the code in the cell to add operators to the operator DAG described above. The DAG is then passed to the query optimizer, which will either immediately kick off the execution of interaction critical paths, if they are present in the DAG, or consider all the non-critical operators to determine what to execute next. We discuss optimizations for non-critical operators in Section 7.5.

After the last interaction is executed and the results are returned, the query optimizer will continue executing operators asynchronously until the entire DAG is executed. In the event that an interaction arrives while a non-critical operator is executing, we preempt the execution of the non-critical operator to avoid delaying the execution of the interaction critical path. We discuss optimizations for supporting effective preemption in Section 7.5.

While the kernel executes operators, a garbage collector (GC) is working in the background to uncache results in the operator DAG to control memory consumption. A GC event is triggered when memory consumption is above 80% of the maximum memory allocated to the kernel, at which point the GC inspects the operator DAG to uncache the set of operator results that are the least likely to speed up future queries. We discuss cache management in Section 7.5.

7.5 Optimization Framework

The opportunistic evaluation framework optimizes for interactive latency by deferring to think timethe execution of operators that do not support interactions. The previous section describes how we use simple program analysis to identify the interaction critical path that must be executed to produce the results for an interaction. In this section, we discuss optimizations for minimizing the latency of a given interaction in Section 7.5 and optimizations for minimizing the latency of future interactions by leveraging think timein Section 7.5. We discuss how to model user behavior to anticipate future interactions in Section 7.5.

Optimizing Current Interactions

Given an interaction critical path, we can apply standard database optimizations for single queries to optimize interactive latency. For example, if the interaction operator is `head` (i.e., examining the first K rows), we can perform predicate pushdown to compute only part of the interaction critical path that leads to the top K rows in the final dataframe. The rest can be computed during think timein anticipation of future interactions.

The main challenge for optimizing interactive latency in opportunistic evaluation is the ability to effectively preempt the execution of non-critical operators. This preemption ensures that we avoid increasing the interactive latency due to irrelevant computation. The current implementation of various operators within pandas and other dataframe libraries often involves calling lower-level libraries that cannot be interrupted during their execution. In such cases, the only way to preempt non-critical operators is to abort their execution completely, potentially wasting a great deal of progress. We propose to overcome this challenge by partitioning the dataframe so that preemptions lead to, in the worst case scenario, only loss of the progress on the current partition.

Dataframe partitioning Partitioning the dataframe in the opportunistic evaluation setting involves navigating the trade-off between the increase in future interactive latencies due to loss of progress during preemption and the reduction in operator latency due to missed holistic optimizations on the entire dataframe. In the setting where interactions are sparse, a single partition maximizes the benefit of holistic optimization while losing progress on the entire operator only occasionally due to preemption. On the other hand, if interactions are frequent and erratic, a large number of small partitions ensures progress checkpointing, at the expense of longer total execution time across all partitions. Thus, the optimal partitioning strategy is highly dependent on user behavior. We discuss how to model user behavior in Section 7.5.

Without a high-fidelity user interaction model, we can create unevenly sized partitions to handle the variability in the arrival rate of interactions. First, we create small partitions for the top and bottom K rows in the dataframe not only to handle the rapid succession of interactions but also to support partial-result queries involving `head` and `tail` that are prevalent in interactive dataframe workloads. Then, for the middle section of the dataframe, the partitions can reflect the distribution of think timesuch that the partition sizes are smaller at intervals where interactions are likely to be issued. For example, if the median think timeis 20s and the operator's estimated execution time is 40s, it might be desirable to have smaller partitions after 50% of the rows have been processed.

The above strategy assumes sequential processing of every row in the dataframe. If, instead, the prevalent workload is working with a select subset of rows, then it is more effective to partition based on the value of the attributes that are commonly used for selection. Of course, partitioning is not necessary if computation started during think timedoes not block computation for supporting interactions.

Note that another important consideration in generating partial results is the selectivity of the underlying operators and whether they are blocking operators. For the former, we may need to employ a much larger partition simply to generate K results. For the latter, we may need to prioritize the generation of the aggregates corresponding to the groups in the top or bottom K (in the case of group-by), or to employ algorithms that prioritize the generation of the K first sorted results (in the case of sorting). In either case, the problem becomes a lot more challenging.

Optimizing Future Interactions Leveraging Think Time

Non-critical Operator scheduling. We now discuss scheduling non-critical operators. Recall that these operators are organized in a DAG built from queries. The job of our scheduler is to decide which *source* operators to execute. Source operators in the DAG are those whose precedent operators do not exist or are already executed. We assume an equal probability of users selecting any operator in the DAG to extend with an interaction.

The scheduler is optimized to reduce the interaction latency; we introduce the notion of an operator's *delivery cost* as the proxy for it. If an operator has not been executed yet, its delivery cost is the cost of executing the operator along with all of its unexecuted predecessors. Otherwise, the delivery cost is zero. Our scheduler prioritizes scheduling the source operator that can reduce the delivery cost across all operators the most. We define a utility function $U(s_i)$ to estimate the benefit of executing a source operator s_i. This function, for a node s_i is set to be the sum of the delivery cost for the source operator and all of its successors D_i:

$$U(s_i) = \sum_{j \in D_i} c_j \tag{7.1}$$

where c_j is the delivery cost for an operator j. Our scheduler chooses to execute the one with the highest $U(s_i)$. This metric prioritizes those operators that "influence" as many expensive downstream operators as possible.

Caching for reuse. When we are executing operators in the background, we store the result of each newly computed operator in memory. However, if the available memory (i.e., the memory budget) is not sufficient to store the new result, we need to recover enough memory by discarding materialized results of previously computed operators. If the discarded materialized results are needed by future operators, we will execute the corresponding operators to recompute them. Here, the optimization problem is to determine which materialized results should be discarded given the memory budget. Our system addresses this problem by systematically considering three aspects of a materialized result, denoted r_i: 1) the chance of r_i being reused, p_i, 2) the cost of recomputing the materialized result, k_i, and 3) the amount of memory it consumes, m_i. We estimate p_i by borrowing ideas from the LRU replacement algorithm. We maintain a counter T to indicate the last time any materialized result is reused and each materialized result is associated with a variable t_i that tracks the last time it is reused. If one materialized result r_i is reused, we increment the counter T by one and set t_i to T. We use the following formula to estimate p_i:

$$p_i = \frac{1}{T + 1 - t_i} \tag{7.2}$$

We see that the more recently a materialized result r_i is reused, the higher p_i is. We can use a cost model as in relational databases to estimate the recomputation cost k_i. We note that we do not always recompute an operator from scratch. Given that the other materialized results are in memory, our cost model estimates the recomputation cost by considering reusing existing materialized results. Therefore, we use the following utility function to decide which materialized result should be discarded.

$$O(r_i) = p_i \times \frac{m_i}{k_i} \tag{7.3}$$

Here, $\frac{m_i}{k_i}$ represents the amount of memory we can spare per unit of recomputation cost to pay. The lower $\frac{m_i}{k_i}$ is, the more likely we discard r_i. Finally, our algorithm will discard the r_i with the lowest $O(r_i)$ value.

Speculative materialization. Our system not only considers caching results generated by users' programs, but also speculatively materializes and caches results that go beyond what users specify, to be used by future operators. One scenario we observed is that users intend to explore the data by changing the value of a filter repeatedly. In this case, we can materialize the intermediate output results before we apply the filter and when users modify the filter, we can reuse the saved results without computing them from scratch. The downside of this approach is that it can increase the latency of computing an interaction when the think timeis limited. Therefore, we enable this optimization only when users' predicted think timeof writing a new operator is larger than the time of materializing the intermediate states.

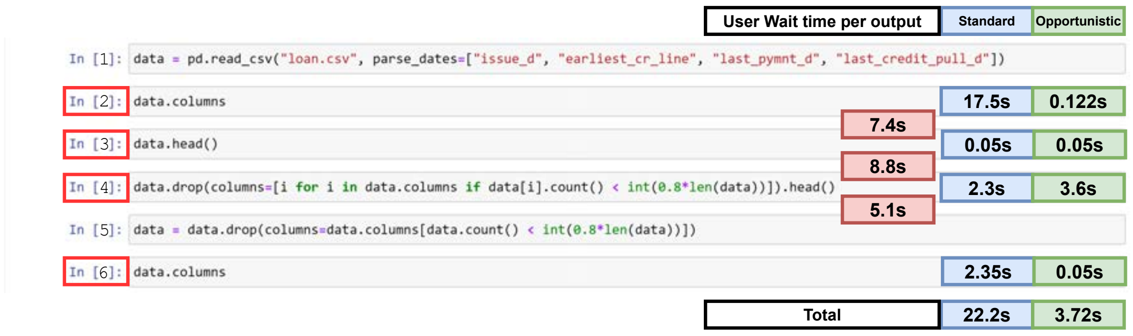
Prediction of User Behavior

The accurate prediction of user behavior can greatly improve the efficacy of opportunistic evaluation. Specifically, we need to predict two types of user behavior: think timeand future interactions. Section 7.3 described some preliminary statistics that can be used to construct a prior distribution for think time. As the system observes the user work, this distribution can be updated to better capture the behavior of the specific user, as we expect the distribution of think timeto vary greatly based on the dataset, task, user expertise, and other idiosyncrasies. These workload characteristics can be factored into the think timemodel for more accurate prediction. This think timemodel can be used by the optimizer to decide the size of dataframe partitions to minimize progress loss due to preemption or to schedule non-critical operators whose expected execution times are compatible with the think timeduration.

To predict future interactions, we can use the models from Yan et al. [129]. These models are trained on a large corpus of data science notebooks from Github. Since future interactions often build on existing operators, we can use the future interaction prediction model to estimate the probabilities of non-critical operators in the DAG leading to future interactions, which can be used by the scheduler to pick non-critical operators to execute next. Let p_j be the probability of the children of an operator j being an interaction. We can incorporate p_j into the utility function in Equation 7.1 to obtain the updated utility function:

$$U_p(s_i) = \sum_{j \in D_i} c_j \times p_j \tag{7.4}$$

Of course, the benefits of opportunistic evaluation can lead to modifications in user behavior. For example, without opportunistic evaluation, a conscientious user might self-optimize by avoiding specifying expensive non-critical operators before interactions, potentially at the cost of code readability. When self-optimization is no longer necessary when authoring queries, the user may choose to group similar operators for better code readability and maintenance, thus creating more opportunities for opportunistic evaluation optimizations.



```
In [1]: data = pd.read_csv("loan.csv", parse_dates=["issue_d", "earliest_cr_line", "last_pymnt_d", "last_credit_pull_d"])
In [2]: data.columns
In [3]: data.head()
In [4]: data.drop(columns=[i for i in data.columns if data[i].count() < int(0.8*len(data))]).head()
In [5]: data = data.drop(columns=data.columns[data.count() < int(0.8*len(data))])
In [6]: data.columns
```

User Wait time per output	Standard	Opportunistic
In [2]	17.5s	0.122s
In [3]	0.05s	0.05s
In [4]	2.3s	3.6s
In [6]	2.35s	0.05s
Total	22.2s	3.72s

Figure 7.9: An example notebook. Cells that show an output are indicated with a red box.

7.6 Case Study

In this section, we evaluate how opportunistic evaluation will impact the end user through a case study. Figure 7.9 shows an excerpt from the original notebook, taken from a Kaggle competition (https://www.kaggle.com/c/home-credit-default-risk).

In this case study, the data scientist first read in the file, and was forced to immediately wait. Then, the user wanted to see the columns that exist in the dataset. This is often done when the data scientist first encounters a dataset. They therefore printed the first 5 lines with `data.head()`. This inspection is important for data validation: the data scientist wanted to ensure that the data was parsed correctly during data ingestion. After these two data validation steps, the data scientist noticed that there were a significant number of `null` values in multiple columns.

The cell labeled `In[4]` shows how the data scientist solved the `null` values problem: they decided to drop any column that does not have at least 80% of its values present. Notice that the data scientist first wanted to see what the results of the query would look like before they executed it, so they added a `.head()` to the end of the query that drops the columns. Likely this was done during debugging, where many different, but similar queries were attempted until the desired output was achieved. The query was then repeated to overwrite the `data` variable. An important note here is that the full dataset is lost at this point due to the overwriting of the `data` variable. The data scientist will need to reread the file if they want access to the full dataset again. After dropping columns with less than 80% of their values present, the data scientist double-checked their work by inspecting the `columns` of the overwritten `data` dataframe. Next, we evaluate the benefits of the opportunistic evaluation approach by determining the amount of synchronous wait time saved by leveraging think time.

To evaluate opportunistic evaluation in our case study, think timewas injected into the notebook from the distribution presented in Figure 7.3. We found that the time that the hypothetical data scientist spent waiting on computation was almost none: the `read_csv` phase took 18.5 seconds originally, but since the output of the `columns` and `head` were prioritized, they were displayed almost immediately (122ms). The data scientist then looked at the two outputs from `columns` and `head` for a combined 16.2 seconds. This means the data scientist synchronously waited on the `read_csv` for approximately 1.3 seconds. Next, the user had to wait another 2.3 seconds for the columns with less than 80% of their values present to be dropped. Without opportunistic evaluation, the user would have to pay this time twice, once to see the first 5 lines with `head` and again to see the `data.columns` output in cell `In[6]`.

7.7 Related Work

As we have discussed in other chapters, many open source systems attempt to provide improved performance or scalability for dataframes. Often, this means only supporting dataframe functionalities that are simple to parallelize (e.g., Dask [32]), or supporting only those operations which can be represented in SQL (e.g. Koalas [63] and SparkSQL [88]). Our project, Modin [73], is the only open source system with an architecture that can support all dataframe operators.

In the research community, there are multiple notable papers that have tackled dataframe optimization through vastly different approaches. Sinthong et al. propose AFrame, a dataframe system implemented on top of AsterixDB by translating dataframe APIs into SQL++ queries that are supported by AsterixDB [107]. Another work by Yan et al. aims to accelerate EDA with dataframes by "auto-suggesting" data exploration operations [129]. Their approach has achieved considerable success in predicting the operations that were actually carried out by users given an observed sequence of operations. More recently, Hagedorn et. al. designed a system for translating pandas operations to SQL and executing on existing RDBMSs [45]. In a similar vein, Jindal et. al. built a system called Magpie for determining the optimal RDBMS to execute a given query [55]. Finally, Sioulas et. al. describe techniques for combining the techniques from recommendation systems to speculatively execute dataframe queries [108].

Our proposed approach draws upon a number of well established techniques from the systems, PL, and DB communities. Specifically, determining and manipulating the DAG of operators blends control flow and data flow analysis techniques from the PL community [25]. The optimization of dataframe operators draws inspiration from battle-tested database approaches such as predicate pushdown, operator reordering, multi-query optimization, and materialized views [48], as well as compiler optimizations such as program slicing and common subexpression elimination. Furthermore, we borrow from the systems literature on task scheduling to take enable asynchronous execution of dataframe operators during think time.

7.8 Conclusion & Future Work

We proposed opportunistic evaluation, a framework for accelerating interactions with dataframes. Interactive latency is critical for iterative, human-in-the-loop dataframe workloads for supporting data validation, both for ML and for EDA. Opportunistic evaluation significantly reduces interactive latency by 1) prioritizing computation directly relevant to the interactions and 2) leveraging think timefor asynchronous background computation for non-critical operators that might be relevant to future interactions. We have shown, through empirical analysis, that current user behavior presents ample opportunities for optimization, and the solutions we propose effectively harness such opportunities.

While opportunistic evaluation addresses data validation prior to model training, data validation challenges are present in other parts of the end-to-end ML workflow. For example, after a trained model has been deployed, it is crucial to monitor and validate online data against the training data in order to detect data drift, both in terms of distribution shift and schema changes. A common practice to address data drift is to retrain the model on newly observed data, thus introducing data drift into the data pre-processing stage of the end-to-end ML workflow. Being able to adapt the data validation steps in a continuous deployment setting to unexpected data changes is an open challenge.

Chapter 8

Open source project community building

Many grad students and professors have asked me for suggestions on how to build a functioning and thriving open source community while in grad school. I thought it would be good to include a chapter in my thesis on how to build an open source community given the success of my open source grad project.

This chapter is part history, part lessons, part advice. I don't know everything, and the moderate success of my work does not mean that my advice is automatically good. These suggestions are based on my experience, so do not take them as absolute truth.

8.1 My History Building a Successful Open Source Project

During my grad school career, I built Modin (https://github.com/modin-project/modin), a full dataframe implementation that, as of writing, has over 6,000 GitHub stars. This effort has been supported by many people over the last few years, so to take all the credit myself would not be fair to those people. Berkeley is well known for creating some of the most used and most impactful software on the planet, so I wasn't starting from nothing.

My approach toward promoting the work has been fairly successful, however I attribute that largely to luck. The first blog post I published (2018) got a lot of input and feedback from others working on the project at the time. It ended up getting shared on Twitter and HackerNews by many people (I had accounts with neither at the time) and generated a lot of interest. At the time, pandas on Ray (which would become Modin) was a 1 month hack I put together with help from several undergraduate students at Berkeley. It honestly wasn't ready for the overwhelming interest it received, and yet it has continued to be developed and grown into something that I couldn't have imagined at the time.

8.2 Lessons and Advice

This section is likely to be long and difficult to parse, so I'm going to make my advice section headers so it's easier to skim to find the points you'd like to better understand.

[1] Make your system understandable to your target user, and don't worry about anyone else

This point is something I think we've gotten right from the beginning with Modin. From the start, we have abstracted away complex details from the user, including in how we present the system. This has, of course, led many of highly technical people to discount the complexity of abstracting away these details, but that has never bothered me. I don't care if someone thinks Modin is or isn't technically interesting, I care that it solves a problem. Because of how I talk about Modin, most people have a simple understanding of the system. That is by design. While working on Modin, we have formalized a new data model, created a new dataframe algebra, and created a truly unique data layout and metadata management system. The people who could understand enough about the underlying system to appreciate it likely wouldn't use Modin in the first place, because Modin is targeting a less-technical group of users. I think this is really important because when people talk about Modin, they generally focus on the problems it solves rather than the technically interesting parts of the implementation. I am okay with that, but make sure that you are. Do you want people to use your work or do you want them to think you're really smart? Sometimes you can have both, but often not.

[2] Be prepared to defer your graduation and publications

This point is less applicable if you have a large team managing the open source, but in my case I was working mostly alone from the open source side. On the research side, we were able to bring together some of the best in databases and machine learning, but many deadlines were missed because of things that came up in the open source. I prioritized the open source community and development over my own graduation and publications. This is a decision you'll have to make for your own situation. I'm not completely convinced now that you need to do this, but at the time I felt like it was necessary to keep the open source community alive and growing. There's little overlap between open source community development and grad school requirements. You are going to have to respond to questions, issues, and promote your work.

[3] The fun parts of open source are front-loaded

At the beginning of any open source project you're going to be able to move fast. There's no technical debt, no new issues, and a lot of energy and excitement. As time goes on, your time will go from developing new features and building things to answering issues and emails. If you're fortunate enough to have a lot of external contributors like Modin does, you'll end up spending a lot of time reviewing code. These days, I spend maybe 20% of my time writing code and debugging. If you want to get into open source, be prepared to spend a large chunk of time on user issues and support after your project hits a critical mass. If you are mostly doing it alone, the project momentum can easily screech to a halt and feel like it's not moving anywhere for weeks. My recommendations here are to (1) avoid romanticizing the idea of managing a highly visible open source project and (2) learn to bin your time. At first, it's easy to answer issues as they come in, eventually you will not get anything done if you always answer issues immediately.

[4] Promotion is important

If you want people to use your project, you have to tell them about it. Part of the difficulty in deciding how to promote is around deciding when. Promotion takes away from development in a small team, and so there needs to be some good reason to promote. Early on, I had planned on curating a series of blog posts that could archive the journey. That was quickly thrown out after the reception from the first blog post. My main advice is to be careful not to overpromote: each update should be substantial. This is mostly personal taste, but I don't like reading a lot of fluffy blogs that have hardly any new content. Honestly, most people aren't going to read the blog anyway, they will skim the headers or scroll to the bottom to read the conclusion. In terms of promotion, getting multiple friendly people to tweet about your blog is probably the best way to promote. HackerNews is not what I would consider a good distribution channel, rather a good place for discussion about topics surrounding a blog's title and content. Podcasts are another way to spread the word, and they are a common way that people hear about new projects.

[5] Make your work easy to install and use

The biggest hurdle to using something is getting started, and the lower the barrier the more people will try it. This might seem obvious but easy means different things to different people so I'll try to be concrete here. Do you require users to pull your Docker container?

Do you require users to build from source? Does installation change the user's environment? Does installation take more than a couple of steps? If you answered yes to any of these questions you're going to have a hard time getting people to do more than look at your README. Making something easy to use is important to getting a community off the ground, if people don't use it they won't tell their friends about it (probably).

The second component to easy use is examples: you need really good useful examples, not toys. Users want to see what they can do with your tool, and showing them how to do a trivial map over a list of integers is not going to give users a good idea of what they can do. Your examples should show off a variety of use cases and capabilities on actual workloads. Examples overlap a bit with the next point on documentation.

[6] Write documentation

Everyone says this, nobody does it.

[7] Primarily use communication channels that are Googleable

Slack, Gitter, etc. are not internet searchable. When people have a problem they will often go to Google first to see if others have solved the problem. In Modin, we use GitHub issues and Discourse boards for discussions to make sure that people looking for solutions can find them. This also has the nice side effect of being able to point to those pages when someone asks the same question.

[8] Give talks at small venues and meetups, not just the big ones

Promoting work via talks at big venues does give you more visibility, but ultimately the smaller more intimate venues are where you're more likely to make good connections with people who will actually use your project. It's tempting at the beginning to try to go straight for giving talks at the big international conferences, but I've found that small meetups are both more focused (people are more likely to have the problem you're solving) and more willing to talk. I gave talks to meetup groups as small as 6 people, and in those meetups I had more engaging conversations that in the larger venues. You need these relationships with your users early on to build momentum. Otherwise, once you do talk at these large venues and people will ask "Who is using your project?". In the large conferences people claim to be looking for the next big technology to adopt, but really they just want to use what everyone else is using. Having users will get you more users, but you need the early adopters, and often you will meet them in small venues or meetups.

[9] Make it easy to reach you

This point will contradict with the point about communication channels being Googleable, but in general you need to be able to be reached by people who run into problems. Don't make the communication overhead of reporting a bug a barrier to discovering the bug exists. In Modin, for example, we set up a couple of emails and if something went wrong internally we asked them to email us a bug report as a part of the error message. It has been pretty successful and there are several bugs we found that weren't discoverable otherwise. We rarely get these today, which is a good indication that things are getting more stable.

Generally, to solve the issue of search indexability, I will ask people who email to open an issue if I triage it to be serious and new. I've found people are easier to go back/forth with over email, so you can get the simple stuff out of the way quickly as well (e.g. user environment issues or user errors).

[10] Scale your efforts later, build a community first

A lot of what I propose here doesn't scale, and it will get worse as the number of users grows. This is by design. Even after you have a critical mass, it's not likely you'll have a large group of contributors outside of your organization (it took roughly 3 years for Modin to get serious outside contributor groups). Building a community is largely a social effort, and you need broadcasting on Twitter is not going to be enough to get the ball rolling. Everyone is doing that. If you want to actually build a community, you need to do things like talk to individuals and answer individual emails. The personal connections are more important than trying to make noise in a very noisy world, and they will get you farther than clicks or views on your tweets and blog posts.

[11] Make it easy to contribute and ask for contributions

Making your project easy to contribute to is a good way to help build a community. There are always people who are interested in working on projects on the side, and getting these people involved is important. Often, projects are too difficult to jump into years later. It is difficult to build a community when you're the only person who knows how to do anything. You're going to need people who can help with issues so you can take some time off every now and then to recharge. This is obvious, but it takes good design and a lot of DevOps work, which doesn't necessarily equate to more code being output. In fact, often helping others will often reduce your own productivity and the overall code velocity of the project. This code velocity cost (on an individual basis) may actually never be recouped, but I argue that there are intangible benefits to working with other people:

- You need to justify your designs

- Producing code is not a good metric of productivity–there are more important things than new features
- Excited contributors often also become evangelists

Adding contributors will not always yield more code or more bugfixes, but it helps build a community.

8.3 Concluding thoughts

I hope this has been helpful. It's a lot of work to keep a community going, and the work is mostly social. There's a lot of engineering in building something, but actually getting the word out and keeping in contact with users takes significant effort.

This list is by no means complete, but I hope it's enough to help you get off the ground. Grad school is a great time to explore a bunch of different things, including open source community building and project management. I hope you can be as successful as I was (or more!) and that this list can help you plan how to execute on your great ideas. Please don't hesitate to reach out to me directly if you have any questions. I don't consider myself an expert on open source community building, but I will do my best to answer any questions you might have.

Chapter 9

Conclusion

This dissertation takes a ground-up approach to implementing scalable dataframe systems. We explored the dataframe data model, designed a system architecture, and created an open-source reference dataframe implementation. In this chapter, we summarize the key findings of this dissertation and outline future directions for dataframe research.

9.1 Summary of Findings

We now detail the findings of this dissertation, labeled Findings ①-②.

Finding ①: A definition for a formal dataframe data model and comprehensive algebra.

In Chapter 3, we proposed a novel dataframe data model and algebra that generalized existing dataframe implementations like pandas [81] and the `data.frame` in R [90]. We focused on pandas because it is by far the most popular dataframe library, but suffers from a lack of scalability. We revisit the dataframe data model figure from Chapter 3 in Figure 9.1.

The dataframe data model consists of four components, each with a set of properties we will discuss here. The figure component labeled A_{mn} is an mXn array that contains the data. R_m is a vector of row labels. Row labels are used to identify a row, but are not necessarily unique as with an index in a database. The row labels originate from within the data itself, meaning dataframe metadata and data are interchangeable. The column labels, C_n in the figure, are a vector of identifiers for columns, and are also not necessarily unique. Rows and columns are interchangeable in a dataframe, so properties that are true for rows must also be true for columns, with the exception of the existence of column types (D_n in Figure 9.1). Columns have types, but rows do not.

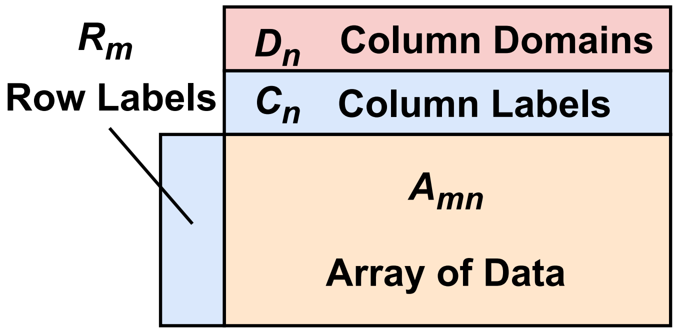


Figure 9.1: The Dataframe Data Model. A copy of this figure can also be found in Chapter 3.

Concretely, dataframes are an abstraction distinct from both relational databases and 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 of any type	Column names from att [3]
Column/row symmetry	Columns and rows are distinct
Support for linear alg. operators	No native support

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

These two equivalent viewpoints are exploited to define a set of algebraic operators for dataframes.

In Chapter 3, we also defined a set of algebraic operators that encompasses the entire pandas API. In Chapter 4, we proved by exhaustion that these operators are sufficient to cover the entire pandas API, and are in fact a generalization of pandas in that we can express even more than pandas with this algebra. The algebraic operators we presented span relational operators (e.g., `join` and `group by`), matrix operators (e.g., `transpose`),

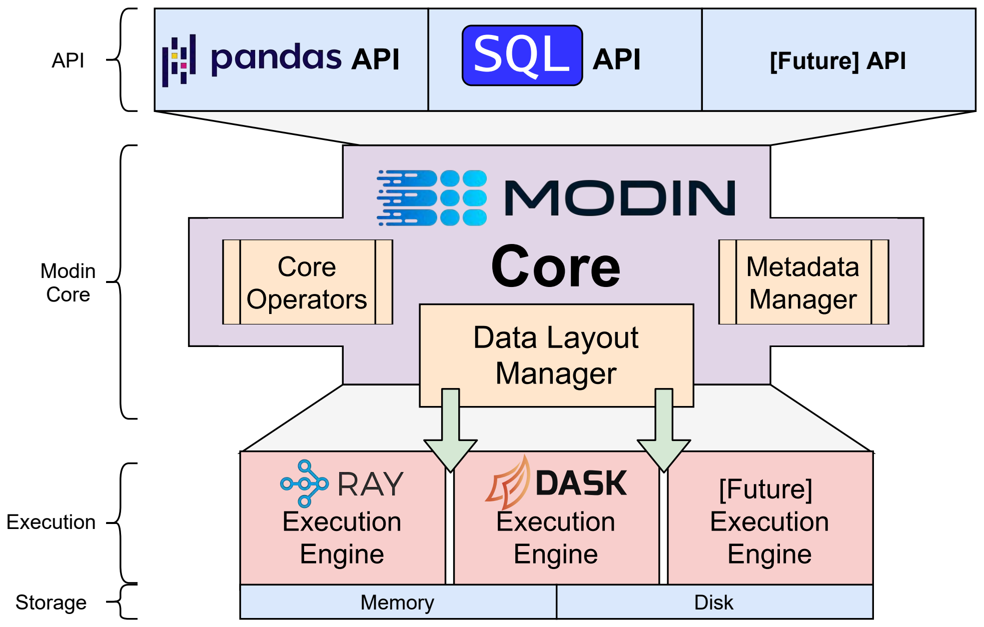


Figure 9.2: The Modin architecture. This figure also appears in Chapter 5.

and dataframe metadata operators (e.g., `infer types` and `from labels`). Collectively, these operators provide significant flexibility to the user to carry out operations on dataframes. The algebra, however, is not a suitable user-facing interface, so we explored an implementation of the data model and algebra in Finding (2).

Finding (2): A scalable implementation of the dataframe.

In Chapter 4, we presented a general dataframe architecture design based on the dataframe data model and algebra presented in Chapter 3. This design includes a description of each operator and the nuances in the behavior. The architecture described in Chapter 4 is scalable by design, and includes possible solutions to key challenges inherent in the dataframe data model.

In Chapter 5, we presented Modin, our reference dataframe implementation and the rules we use to determine the optimal parallelism in this implementation. The architecture of Modin is shown in Figure 9.2 and is designed to support multiple execution backends. This design decision stems from the goal of supporting data science in all environments, such that data scientists can use the same notebook with Modin in different operating environments with the same results. Modin exposes the familiar pandas API to users by translating pandas operators into the underlying algebra implementation that we discuss in Chapters 3 and 4. We also discussed low level implementation decisions, with an additional focus on how we manage metadata in Modin. Each operator manipulates the metadata in specific ways, which we explained in Chapter 5. We discussed how we overcome many of the challenges in having a distributed implementation of a dataframe, including keeping certain metadata close to the user (e.g., row and column labels) while also trying to keep the labels close to the data for when they are inevitably operated on as data. Modin's impact in the open source community can be measured by the number of GitHub stars (6,000+) and installs (over one million) [73]. The impact of Modin shows that there is a significant need for distributed dataframe implementations.

9.2 Future Work

Given how recent the dataframe data model and algebra are, dataframe research remains an interesting an open area with many interesting directions. The future work here was initially introduced in prior chapters where it is also contextualized.

Speculative Query Execution

Much of the work presented in this dissertation focuses on making the data scientist, i.e. the human user, more efficient overall. Speculative query execution is yet another way to improve the user's overall productivity. By accurately executing queries before the user submits them, we can decrease the amount of time the user waits for the query result. One major challenge in designing a speculative query execution engine is accurately predicting what the user might do. The user model would likely need to adapt as the data scientist uses the system to become more accurate. If the speculation is incorrect, it is important that the user is not penalized. This may mean throwing away partial results for queries that are never needed, but the highest priority should always be improving the user's productivity by decreasing the amount of time the user spends waiting on results.

Query Optimization

The new dataframe algebra provides a lot of interesting future work in designing query optimizers that can efficiently rearrange the order of operators. Our focus with the algebra was on expressiveness and minimality, so that any one operation has only one way of being expressed. In addition to query rewriting and reordering, optimizing the physical layout of the data remains an interesting open challenge. Avoiding data shuffling and expensive transpose operations will be key to the scalability of dataframes.

Additionally, multi-query optimization (MQO) through partial result materialization and reuse should provide a significant improvement to existing users. During a typical dataframe workflow, users will commonly re-execute the same code or code with very small changes. For expensive queries, we can avoid recomputing results to provide an improved user experiences. If the user makes minor changes, it may be possible to reuse the some or all of the intermediate results of original query instead of recomputing the entire query. For example, if the user changes a `map` operator at the end of a long query to add one instead of two, we can simply subtract one from the original result in a `map` instead of rerunning the entire query from the beginning. The extent to which MQO is possible in the new algebra is not yet known, but this is likely to be an area that can significantly improve the overall performance of the system.

9.3 Final Remarks

Dataframes have become universally popular as a means to flexibly represent data in various stages of structure, and manipulate it using a rich set of operators—thereby becoming an essential tool in data scientists' toolbox. However, popular dataframe systems like pandas scale poorly—and are non-interactive even on moderate to large datasets. In this dissertation, we take a ground-up approach to improving the scalability and performance of dataframes. We began with a formalism to ground the discussion in theory. We showed that dataframes can not only be well defined, but have a unique set of features that make it difficult to scale. We discussed our experiences developing Modin, our reference parallel dataframe system, which already has users across several industries, and considerable traction within the open source GitHub community with over 1M downloads. Modin translates pandas functions into a core set of operators that are individually parallelized via a set of columnar, row-wise, and cell-wise decomposition rules that we formalized in this dissertation. With careful engineering, a dataframe should empower the data scientist without new requirements. Looking forward, dataframes will continue to be important structures for ad-hoc and exploratory data analysis, and dataframe systems will need to continue to be improved and developed to enhance the productivity of the data scientist.

Bibliography

[1] Daniel Abadi et al. "The design and implementation of modern column-oriented database systems". In: *Foundations and Trends® in Databases* 5.3 (2013), pp. 197–280.

[2] Martın Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Vol. 8. Addison-Wesley Reading, 1995.

[4] Azza Abouzeid et al. "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads". In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 922–933.

[5] Sameer Agarwal et al. "BlinkDB: queries with bounded errors and bounded response times on very large data". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 29–42.

[6] Josep Aguilar-Saborit and Raghu Ramakrishnan. "POLARIS: The Distributed SQL Engine in Azure Synapse". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3204–3216. DOI: `10.14778/3415478.3415545`. URL: `http://www.vldb.org/pvldb/vol13/p3204-saborit.pdf`.

[7] Daniel Alabi and Eugene Wu. "Pfunk-h: Approximate query processing using perceptual models". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM. 2016, p. 10.

[8] Abdallah M Alashqur, Stanley YW Su, and Herman Lam. "OQL: a query language for manipulating object-oriented databases". In: *Proceedings of the 15th international conference on Very large data bases*. Morgan Kaufmann Publishers Inc. 1989, pp. 433–442.

[9] Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.

[10] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1383–1394. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742797. URL: http://doi.acm.org/10.1145/2723372.2742797.

[11] Leilani Battle and Jeffrey Heer. "Characterizing Exploratory Visual Analysis: A Literature Review and Evaluation of Analytic Provenance in Tableau". In: *Eurographics Conference on Visualization (EuroVis) 2019* 38.3 (2019).

[12] Michael Bayer. "SQLAlchemy". In: *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. Ed. by Amy Brown and Greg Wilson. aosabook.org, 2012. URL: http://aosabook.org/en/sqlalchemy.html.

[13] Mangesh Bendre et al. "Dataspread: Unifying databases and spreadsheets". In: *PVLDB* 8.12 (2015), pp. 2000–2003.

[14] Mangesh Bendre et al. "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)*. IEEE. 2018, pp. 113–124.

[15] MKABV Bittorf et al. "Impala: A modern, open-source sql engine for hadoop". In: *Proceedings of the 7th biennial conference on innovative data systems research*. 2015.

[16] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". In: *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005, pp. 225–237. URL: http://cidrdb.org/cidr2005/papers/P19.pdf.

[17] Umit V Catalyurek and Cevdet Aykanat. "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication". In: *IEEE Transactions on parallel and distributed systems* 10.7 (1999), pp. 673–693.

[18] Don Chamberlin et al. "XQuery 1.0: An XML query language". In: *W3C working draft* 7 (2001).

[19] J. Chambers, T. Hastie, and D. Pregibon. "Statistical Models in S". In: *Compstat*. Ed. by Konstantin Momirović and Vesna Mildner. Heidelberg: Physica-Verlag HD, 1990, pp. 317–321. ISBN: 978-3-642-50096-1.

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

[21] Fay Chang et al. “Bigtable: A distributed storage system for structured data”. In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26.

[22] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. " O’Reilly Media, Inc.", 2013.

[23] Jaeyoung Choi et al. “ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers”. In: *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE. 1992, pp. 120–127.

[24] *ClickHouse is a fast open-source OLAP database management system*. 2021. URL: `https://clickhouse.tech/`.

[25] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[26] Conor Cunningham, César A Galindo-Legaria, and Goetz 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*. VLDB Endowment. 2004, pp. 998–1009.

[27] Joseph Vinish D’silva, Florestan De Moor, and Bettina Kemme. “AIDA - Abstraction for Advanced In-Database Analytics”. In: *PVLDB* 11 (2018), pp. 1400–1413.

[28] Benoit Dageville et al. “The snowflake elastic data warehouse”. In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 215–226.

[29] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[30] *Dask DataFrame API Reference*. `https://docs.dask.org/en/latest/dataframe-api.html`. 2021.

[31] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: `https://dask.org`.

[32] *Dask Documentation*. `https://docs.dask.org/en/latest/`. 2020.

[33] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[34] Bolin Ding et al. “Sample+ seek: Approximating aggregates with distribution precision guarantee”. In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 679–694.

[35] Leonidas Fegaras. “Query unnesting in object-oriented databases”. In: *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 1998, pp. 49–60.

[36] Philippe Flajolet et al. “Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm”. In: 2007.

[37] Edgar Gabriel et al. “Open MPI: Goals, concept, and design of a next generation MPI implementation”. In: *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer. 2004, pp. 97–104.

[38] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. “SharedDB: killing one thousand queries with one stone”. In: *PVLDB* 5.6 (2012), pp. 526–537.

[39] Roy Goldman and Jennifer Widom. “WSQ/DSQ: A practical approach for combined querying of databases and the web”. In: *ACM SIGMOD Record*. Vol. 29. 2. ACM. 2000, pp. 285–296.

[40] Gene H Golub and Charles F Van Loan. *Matrix computations*. Vol. 3. JHU press, 2013.

[41] *Google Colab*. `colab.research.google.com`.

[42] Goetz Graefe. “Volcano/spl minus/an extensible and parallel query evaluation system”. In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (1994), pp. 120–135.

[43] Goetz Graefe, Ross Bunker, and Shaun Cooper. “Hash joins and hash teams in Microsoft SQL Server”. In: *VLDB*. Vol. 98. Citeseer. 1998, pp. 86–97.

[44] Anurag Gupta et al. “Amazon Redshift and the Case for Simpler Data Warehouses”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 1917–1923. DOI: `10.1145/2723372.2742795`. URL: `https://doi.org/10.1145/2723372.2742795`.

[45] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. “Putting Pandas in a Box”. In: *The Conference on Innovative Data Systems Research (CIDR)* ().

[46] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. “QPipe: a simultaneously pipelined relational query engine”. In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM. 2005, pp. 383–394.

[47] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. “Online aggregation”. In: *Acm Sigmod Record*. Vol. 26. 2. ACM. 1997, pp. 171–182.

[48] Joseph M Hellerstein, Michael Stonebraker, James Hamilton, et al. “Architecture of a database system”. In: *Foundations and Trends® in Databases* 1.2 (2007), pp. 141–259.

[49] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*. Vol. 1. Microsoft research Redmond, WA, 2009.

[50] Richard D Hipp. *SQLite*. Version 3.31.1. 2020. URL: `https://www.sqlite.org/index.html`.

[51] Dylan Hutchison, Bill Howe, and Dan Suciu. "Lara: A key-value algebra underlying arrays and relations". In: *arXiv preprint arXiv:1604.03607* (2016).

[52] Dylan Hutchison, Bill Howe, and Dan 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*. ACM. 2017, p. 2.

[53] *Ibis Documentation*. 2021. URL: `https://ibis-project.org/docs/index.html`.

[54] Ross Ihaka and Robert Gentleman. "R: a language for data analysis and graphics". In: *Journal of computational and graphical statistics* 5.3 (1996), pp. 299–314.

[55] Alekh Jindal et al. "Magpie: Python at Speed and Scale using Cloud Backends". In: *The Conference on Innovative Data Systems Research (CIDR)* ().

[56] Manas Joglekar et al. "Exploiting correlations for expensive predicate evaluation". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1183–1198.

[57] Albert Kim et al. "Optimally leveraging density and locality for exploratory browsing and sampling". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM. 2018, p. 7.

[58] Albert Kim et al. "Rapid sampling for visualizations with ordering guarantees". In: *PVLDB* 8.5 (2015), pp. 521–532.

[59] Steffen Kläbe and Stefan Hagedorn. "Applying Machine Learning Models to Scalable DataFrames with Grizzly". In: *BTW 2021* (2021).

[60] Steffen Kläbe and Stefan Hagedorn. "When Bears get Machine Support: Applying Machine Learning Models to Scalable DataFrames with Grizzly". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2021) 13.–17. September 2021 in Dresden, Deutschland* (), p. 195.

[61] Thomas Kluyver et al. "Jupyter Notebooks-a publishing format for reproducible computational workflows." In: *ELPUB*. 2016, pp. 87–90.

[62] Donald Ervin Knuth. *The art of computer programming*. Vol. 3. Pearson Education, 1997.

[63] *Koalas: pandas API on Apache Spark*. `https://koalas.readthedocs.io/en/latest/`. Date accessed: 2019-12-27. 2019.

[64] Laks VS Lakshmanan, Fereidoon Sadri, and Iyer N Subramanian. "SchemaSQL-a language for interoperability in relational multi-database systems". In: *VLDB*. Vol. 96. Citeseer. 1996, pp. 239–250.

[65] Willis Lang et al. "Partial results in database systems". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 1275–1286.

[66] Stephen Macke et al. "Adaptive sampling for rapidly matching histograms". In: *PVLDB* 11.10 (2018), pp. 1262–1275.

[67] Stephen Macke et al. "Fine-Grained Lineage for Safer Notebook Interactions". In: *Proceedings of the VLDB Endowment* (2021).

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

[69] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[70] *Meet the man behind the most important tool in data science*. `https://qz.com/1126615/the-story-of-the-most-important-tool-in-data-science/`. 2017.

[71] Sergey Melnik et al. "Dremel: A Decade of Interactive SQL Analysis at Web Scale". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3461–3472. DOI: `10.14778/3415478.3415568`. URL: `http://www.vldb.org/pvldb/vol13/p3461-melnik.pdf`.

[72] Sergey Melnik et al. "Dremel: interactive analysis of web-scale datasets". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 330–339.

[73] *Modin Documentation*. `https://modin.readthedocs.io/en/latest/`. 2020.

[74] Todd Mostak. "An overview of MapD (massively parallel database)". In: *White paper. Massachusetts Institute of Technology* (2013).

[75] Barzan Mozafari and Ning Niu. "A Handbook for Building an Approximate Query Engine." In: *IEEE Data Eng. Bull.* 38.3 (2015), pp. 3–29.

[76] Gonzalo Navarro and Kunihiko Sadakane. "Fully functional static and dynamic succinct trees". In: *ACM Transactions on Algorithms (TALG)* 10.3 (2014), pp. 1–39.

[77] New York (N.Y.). Taxi And Limousine Commission. *New York City Taxi Trip Data, 2009-2018*. eng. 2019. DOI: `10.3886/icpsr37254.v1`. URL: `https://www.icpsr.umich.edu/icpsrweb/ICPSR/studies/37254/versions/V1`.

[78] Christopher Olston et al. "Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1099–1110.

[79] Ricardo Otazo, Emmanuel Candes, and Daniel K Sodickson. "Low-rank plus sparse matrix decomposition for accelerated dynamic MRI with separation of background and dynamic components". In: *Magnetic resonance in medicine* 73.3 (2015), pp. 1125–1136.

[80] M Tamer Özsu and Patrick Valduriez. "Distributed and parallel database systems". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 125–128.

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

[82] Aditya Ganesh Parameswaran et al. "Deco: declarative crowdsourcing". In: *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM. 2012, pp. 1203–1212.

[83] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. "Visualization-aware sampling for very large databases". In: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE. 2016, pp. 755–766.

[84] Yongjoo Park et al. "VerdictDB: universalizing approximate query processing". In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 1461–1476.

[85] Fernando Perez and Brian E Granger. "Project Jupyter: Computational narratives as the engine of collaborative data science". In: *Retrieved September* 11.207 (2015), p. 108.

[86] Fernando Pérez and Brian E Granger. "IPython: a system for interactive scientific computing". In: *Computing in Science & Engineering* 9.3 (2007).

[87] Devin Petersohn et al. "Towards Scalable Dataframe Systems". In: *arXiv preprint arXiv:2001.00888* (2020).

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

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

[90] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2017. URL: `https://www.R-project.org/`.

[91] Mark Raasveldt and Hannes Mühleisen. "DuckDB: an Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1981–1984.

[92] Vijayshankar Raman and Joseph M Hellerstein. "Partial results for online query processing". In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM. 2002, pp. 275–286.

[93] Vijayshankar Raman and Joseph M Hellerstein. "Potter ' s Wheel : An Interactive Data Cleaning System". In: *Proceedings of the 27th VLDB Conference* (2001).

[94] Vijayshankar Raman, Bhaskaran Raman, and Joseph M Hellerstein. "Online dynamic reordering for interactive data processing". In: *VLDB*. Vol. 99. 1999, pp. 709–720.

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

[96] Matthew Rocklin. "Dask: Parallel computation with blocked algorithms and task scheduling". In: *Proceedings of the 14th Python in Science Conference*. 130-136. Citeseer. 2015.

[97] Lawrence A Rowe and Michael R Stonebraker. "The POSTGRES data model". In: *Readings in object-oriented database systems* (1990), pp. 461–473.

[98] Prasan Roy et al. "Efficient and extensible algorithms for multi query optimization". In: *ACM SIGMOD Record*. Vol. 29. 2. ACM. 2000, pp. 249–260.

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

[100] Adam Rule, Aurélien Tabard, and James D. Hollan. "Exploration and Explanation in Computational Notebooks". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, 32:1–32:12. ISBN: 978-1-4503-5620-6. DOI: `10.1145/3173574.3173606`. URL: `http://doi.acm.org/10.1145/3173574.3173606`.

[101] Florin Rusu and Yu Cheng. "A survey on array storage, query languages, and systems". In: *arXiv preprint arXiv:1302.0103* (2013).

[102] *Scaling to Large Datasets, Pandas Documentation*. https://pandas.pydata.org/pandas-docs/stable/user_guide/scale.html. Date accessed: 2019-12-27. 2019.

[103] P Griffiths Selinger et al. In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM. 1979, pp. 23–34.

[104] Timos K Sellis. “Multiple-query optimization”. In: *ACM Transactions on Database Systems (TODS)* 13.1 (1988), pp. 23–52.

[105] Jayavel Shanmugasundaram et al. “Querying XML views of relational data”. In: *VLDB*. Vol. 1. 2001, pp. 261–270.

[106] Manish Singh, Arnab Nandi, and HV Jagadish. “Skimmer: rapid scrolling of relational query results”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 181–192.

[107] Phanwadee Sinthong and Michael J Carey. “AFrame: Extending DataFrames for Large-Scale Modern Data Analysis (Extended Version)”. In: *arXiv preprint arXiv:1908.06719* (2019).

[108] Panagiotis Sioulas et al. “Accelerating Complex Analytics using Speculation”. In: *The Conference on Innovative Data Systems Research (CIDR)* ().

[109] *Spark DataFrame API Reference*. https://spark.apache.org/docs/1.6.3/api/java/org/apache/spark/sql/DataFrame.html. 2021.

[110] *Stack Overflow*. 2021. URL: http://stackoverflow.com/.

[111] Michael Stonebraker et al. “SciDB: A database management system for applications with complex analytics”. In: *Computing in Science & Engineering* 15.3 (2013), pp. 54–62.

[112] Mike Stonebraker et al. “C-store: a column-oriented DBMS”. In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2018, pp. 491–518.

[113] R Core Team et al. “R: A language and environment for statistical computing”. In: (2013).

[114] *Ten Things I Hate About Pandas*. https://wesmckinney.com/blog/apache-arrow-pandas-internals/. Date accessed: 2019-12-27. 2017.

[115] *Teradata | Data Analytics for a Hybrid Multi-Cloud World*. 2021. URL: https://www.teradata.com/.

[116] Ashish Thusoo et al. “Hive-a petabyte scale data warehouse using hadoop”. In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE. 2010, pp. 996–1005.

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

[118] John W Tukey. *Exploratory data analysis*. Vol. 2. Reading, Mass., 1977.

[119] *Vaex: Out-of-Core DataFrames for Python*. `https://github.com/vaexio/vaex`. Date accessed: 2019-12-27. 2019.

[120] Stratis D Viglas and Jeffrey F Naughton. "Rate-based query optimization for streaming information sources". In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM. 2002, pp. 37–48.

[121] Stratis D Viglas, Jeffrey F Naughton, and Josef 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*. VLDB Endowment. 2003, pp. 285–296.

[122] Mehul Nalin Vora. "Hadoop-HBase for large-scale data". In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*. Vol. 1. IEEE. 2011, pp. 601–605.

[123] Song Wang, Elke A Rundensteiner, and Murali Mani. "Optimization of nested xquery expressions with orderby clauses". In: *Data & Knowledge Engineering* 60.2 (2007), pp. 303–325.

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

[125] Hadley Wickham. "Tidy data". In: *The Journal of Statistical Software* 59 (10 2014). URL: `http://www.jstatsoft.org/v59/i10/`.

[126] Catharine M Wyss and Edward L Robertson. "A formal characterization of PIVOT/UNPIVOT". In: *Proceedings of the 14th ACM international conference on Information and knowledge management*. 2005, pp. 602–608.

[127] Doris Xin et al. "Enhancing the Interactivity of Dataframe Queries by Leveraging Think Time". In: *Bulletin of the Technical Committee on Data Engineering*. Vol. 4. IEEE. 2021.

[128] Doris Xin et al. "Helix: Holistic optimization for accelerating iterative machine learning". In: *PVLDB* 12.4 (2018), pp. 446–460.

[129] Cong Yan and Yeye He. "Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1539–1554.

[130] Kai Zeng et al. “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*. ACM. 2014, pp. 277–288.

[131] Yi Zhang, Herodotos Herodotou, and Jun Yang. “RIOT: I/O-efficient numerical computing without SQL”. In: *arXiv preprint arXiv:0909.1766* (2009).

[132] Tianyi Zhou and Dacheng Tao. “Godec: Randomized low-rank & sparse matrix decomposition in noisy case”. In: *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*. 2011.