Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System

Devin Petersohn*, Dixin Tang*, Rehan Durrani, Areg Melik-Adamyan†, Joseph E. Gonzalez, Anthony D. Joseph, Aditya G. Parameswaran
UC Berkeley | † Intel
{devin.petersohn,totemtang,rdurrani,jegonzal,adj,adityagp}@berkeley.edu,areg.melik-adamyan@intel.com

ABSTRACT

Dataframes have become universally popular as a means to represent data in various stages of structure, and manipulate it using a rich set of operators—thereby becoming an essential tool in the data scientists' toolbox. However, dataframe systems, such as pandas, scale poorly—and are non-interactive on moderate to large datasets. We discuss our experiences developing MODIN, our first cut at a parallel dataframe system, which already has users across several industries and over 1M downloads. MODIN translates pandas functions into a core set of operators that are individually parallelized via columnar, row-wise, or cell-wise decomposition rules that we formalize in this paper. We also introduce metadata independence to allow metadata—such as order and type—to be decoupled from the physical representation and maintained lazily. Using rule-based decomposition and metadata independence, along with careful engineering, MODIN is able to support pandas operations across both rows and columns on very large dataframes—unlike Koalas and Dask DataFrames that either break down or are unable to support such operations, while also being much faster than pandas.

PVLDB Reference Format:
Devin Petersohn, Dixin Tang, Rehan Durrani, Areg Melik-Adamyan, Joseph E. Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System. PVLDB, 15(3): 739-751, 2022.
doi:10.14778/3494124.3494152

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/modin-project/modin.

1 INTRODUCTION

Dataframe systems, such as pandas [5], have been widely embraced by data scientists to perform tasks spanning transformation, validation, cleaning, and exploration. pandas is estimated to have 5-10M users [3], and has been deemed to be "*the most important tool in data science*" [1]. The popularity can be attributed to many factors, including the flexible data model and rich set of functions or operators. From the data model standpoint, dataframes employ a flexible and intuitive tabular data model, with no pre-defined schema and support for mixed types per column, symmetric treatment of rows and columns, and row and column ordering. Data scientists can quickly get started on analysis without having to declare a schema or resolve type issues, and can employ non-relational operations useful in data analysis (such as `transpose`). From the operator standpoint, dataframe systems provide a rich and varied set tailored to data science, allowing users to operate equivalently across both rows and columns; pandas supports over 600 such functions. For example, `fillna` allows data scientists to clean data by filling in `NULL` values, without having to write custom code.

At the same time, 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 [2, 6, 32–34, 42, 45]. 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 (https://github.com/modin-project/modin), a *parallel dataframe system*, acting as a drop-in replacement for pandas. MODIN is already being used by data scientists across industries, including telecom, finance, and automotive, has been ***downloaded more than 1 Million times***, with over 75 contributors across 12+ institutions, and more than 6.4k GitHub stars (as of September 2021). 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 doing so, we make first steps towards the vision we had outlined in our previous paper [42], wherein we proposed a candidate dataframe algebra. In this paper we operationalize and extend this algebra in a real implementation of MODIN, and 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 develop formal *decomposition rules* that allow us to rewrite operations on the original dataframe into analogous operations on vertical, horizontal, or block-based partitions of the dataframe while being able to concatenate the outputs to reproduce the results on the original operations. These decomposition rules respect the unique properties of dataframes, such as preserving ordering and supporting mixed column types. Further, column types may change in the decomposed dataframes in unpredictable ways, requiring possibly expensive coordination across decompositions. Moreover, the flexible data model blurs the boundary between data and metadata, and supports operators that query and manipulate data and metadata

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494152

*Equal contribution

at the same time—identifying decomposition rules for parallelizing such operations is non-trivial. For example, unlike relational databases, dataframes allow elevating data to and from metadata. In addition, the labels, types, and shape of an output dataframe are not just based on the operators, but also depend on the data (e.g., when dropping all columns with NULL values). Dataframe operators commonly mix both data and metadata operations.

Finally, we 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. We draw on our proposed candidate algebra [42], but extend it to make it practical—for example, our prior algebra requires us to repeatedly take transposes to apply columnar operations; here, we natively support columnar versions of operations. Distilling the 600+ functions in a system such as pandas into a small core set of operators posed a substantial engineering challenge.

Metadata Independence. Dataframe systems make several metadata-related design decisions that impact scalability and semantics. In particular, they tightly couple metadata with the physical representation; 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, pandas eagerly determines and materializes the type of each column at the end of each operation—a time-consuming blocking step on large dataframes. Morover, pandas often coerces types when this may not be intended, 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 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 support *order independence* wherein the physical order can match the logical order on demand, but isn't done unless necessary. Overall, ensuring correct type and ordering semantics for dataframe operators is a big challenge.

Our Approach. In this work, we address the scalability and semantics challenges and instantiate our ideas in MODIN. MODIN adopts a small set of core operators (proposed in our vision paper [42]) to implement the wide set of dataframe operations. 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× 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 [32, 33, 45]; 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. We describe other differences with respect to metadata management in Section 7. Koalas [4], Dask [44], and Ibis [12] 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 [28]; however, the matrix data model (with homogenous data types) and set of operators are both very different, necessitating different decomposition rules.

Contributions and Outline. Our contributions are as follows:

- We formalize the notion of flexible *dataframe decompositions* across multiple dimensions, and outline decomposition rules for each of the core operators underlying MODIN—allowing these operators to be executed in parallel. We also introduce strategies for choosing between decomposition rules in MODIN and identify two multi-operator optimization strategies that immediately extend from the decomposition schemes (Section 3).
- We introduce *metadata independence* for dataframes, including a flexible type system for dataframes that enabled deferred and correct inference of types only when needed. We discuss how to decouple logical ordering from physical ordering of dataframes, and a mechanism for dual but lazy maintenance of labels along with and separate from the data to facilitate easy lookup. We describe the ordering and typing aspects for our core dataframe operators (Section 4).
- We describe the physical layout of MODIN and compare it with existing systems, such as array-oriented databases [22, 41] (Section 5).
- We evaluate MODIN against existing systems like Koalas [4] and Dask DataFrame [11], in addition to pandas [5]. We demonstrate speedups of up to 100× over pandas and Koalas, and 50× over Dask DataFrame. We also evaluate the end-to-end performance of MODIN on real applications and demonstrate performance improvements of individual optimization techniques introduced in this paper. Finally, we perform an experiment to show MODIN's performance benefit in a laptop setting (Section 6).

2 BACKGROUND AND PROBLEMS

In this section, we provide a brief recap of the dataframe data model and MODIN's approach from our vision paper [42] for completeness. Then, we discuss the research problems that we focus on in this paper, but are not addressed in the vision paper.

2.1 Background

Dataframe data model. A dataframe D is a tuple (A, R, C, T), where A is an $m \times n$ array of data entries that represents the dataframe

content, R is an array of m row labels, C is an array of n column labels, and T is array of types for each column [42]. Given they are arrays, all of A, R, C, and T are ordered. Dataframe operators either maintain order or modify it based the semantics of the operator. 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.

MODIN **architecture.** The architecture of MODIN is composed of four layers: the API layer, the MODIN core layer, and the execution and storage layer. MODIN's API layer is modular in order to support multiple modes of interaction, including the pandas API, SQL, or Spark DataFrame API [14].

To support these multiple modes, MODIN defines a compact set of powerful and extensible operators that can implement existing APIs and define new ones as part of the MODIN core layer. These operators include i) dataframe versions of relational ones (e.g., `join`), ii) non-relational operators that query and manipulate metadata (e.g., `infer_types` and `transpose`) to support flexible schema and mixed types, and iii) low-level operators (e.g., `map`, `groupby`, and `explode`) that accept an input function. We will describe the semantics of decomposition for these operators in Section 3. While the vision paper [42] introduces the core operators, it does not discuss how to parallelize them and efficiently manage metadata, which will be the focus of this paper. We also modify the core operators to allow for column-oriented versions of these operators (specified as `axis` in pandas) to avoid expensive transposes.

After MODIN decides the approach to parallelizing the core operators, they will be run by underlying execution engines, such as Ray [37] and Dask. MODIN currently defaults to Ray. The Dask engine [44] in MODIN is not to be confused with the Dask Dataframe [11]. MODIN can use Dask's distributed scheduler, but does not share any code with Dask Dataframe.

The storage layer of MODIN decides the storage format for the dataframes. Currently, MODIN adopts the data format of pandas by default, but is flexible enough to support other formats. This layer additionally decides the caching policy for dataframes such that MODIN can support out-of-core computation.

2.2 Research Problems

Here are the research problems we focus on in this paper.

Formal decomposition of dataframe operators. To ensure the scalability of MODIN, we decompose dataframes into smaller partitions, enabling parallel execution on the partitions. Our research problem here is to formally define decomposition rules for the core dataframe operators, so as to maintain ordering, support flexible access patterns (row, column, and cell-wise), and parallelize operators unique to dataframes. We discuss decomposition rules in Section 3.

Metadata management. MODIN has a metadata manager responsible for maintaining metadata, including data types, column and row labels, and the mapping between logical and physical order.

The unique challenge with dataframes is that one column can contain values from one or more types. To find these types, we need to scan the column, which incurs significant overhead. In addition, operators can change type information in data-dependent ways. Our research problem here is to formally define the semantics of mixed

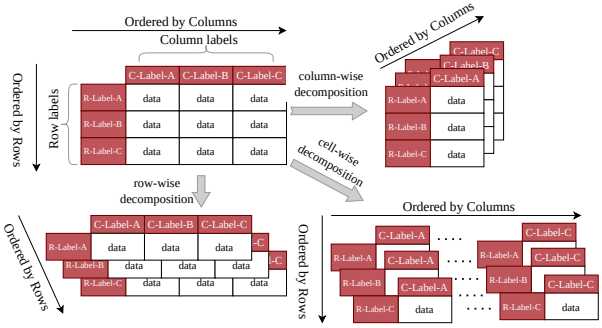


Figure 1: Cell/row/column-wise decomposition

typed-columns and how types are changed across MODIN's core operators, and to reduce the overhead of finding types in dataframes.

Managing row and column labels is also non-trivial because metadata can become data, and vice-versa. For example, row labels may be inserted into the data and operated on as data. In addition to this interchange, users have expectations for low latency interactions when they lookup rows or columns by labels. Therefore, the challenge here is to efficiently support querying and updating the labels at the same time.

Finally, maintaining order is also challenging. We need to define how order is changed across operators, which is not covered in existing systems. In addition, inferring the precise position of each row or column is time-consuming and should not be repeatedly performed after each operator. Therefore, another research problem here is how to defer this costly position inference. We address the aforementioned research problems and challenges in Section 4.

3 DECOMPOSITION & OPTIMIZATION

We formally define the semantics of dataframe decompositions and propose a set of decomposition rules for parallelizing operators over dataframe decompositions.

3.1 Semantics of Dataframe Decomposition

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 1 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. The row-wise and column-wise decomposition decomposes a dataframe into a set of row and column dataframes, respectively. A row dataframe $D_{i*} = (A_{i*}, R_i, C, T)$ appends all of the unit dataframes with the same row labels as new columns in order. 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. Note that unlike the relational context where we 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 1, can be summarized as follows:

$$D = \bigoplus_{i=1}^{m}{}_r \bigoplus_{j=1}^{n}{}_c D_{ij} = \bigoplus_{*} D_{ij} = \bigoplus_{i=1}^{m}{}_r D_{i*} = \bigoplus_{j=1}^{n}{}_c D_{*j}$$

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

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

$filter_r$ selects the rows whose *cols*'s composite key equals k_i and $\bigoplus_{g(cols)}$ appends the groups 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 groups by the groupby key, which is used by the sort operator. We will discuss this decomposition in Section 3.2.3 when we introduce the sort operator.

3.2 Decomposition Rules for Operators

We now describe the decomposition rules for the core operators in MODIN. A core operator often takes a function as input. 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 this is a *user-defined function (UDF)*. Or this function can be in-built into the system by the developer implementing the API in MODIN , e.g., `fillna` in pandas, where `NULL` values are filled in using a specific approach. We call this a *system predefined function (SPF)*.

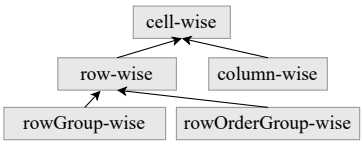


Figure 2: The hierarchy of decompositions: a parent node represents a more general decomposition than its children.

Each decomposition rule uses one or more types of decompositions discussed above. The five types of decomposition form a tree structure (shown in Figure 2) 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 decomposition type because replacing this one with 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.

Rulebox 1: decomposition rules for low-level operators

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

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

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

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

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 4. In the following, we use f to represent a UDF or SPF (system predefined function) while h is used to represent an SPF, as defined early on in Section 3.

3.2.1 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 2, `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 use 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.

`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

[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

$$\texttt{inferT} : inferT(D) = \bigoplus_{j=1}^{n}{}_c\, h_c^{infer}(D_{*j})$$

$$\texttt{filterT} : filterT(D,t) = \bigoplus_{i=1}^{m}{}_r \bigoplus_{j=1}^{n}{}_c\, mask(h_*^{lb}(t,D), D_{ij})$$

$$\texttt{to\_labels} : to\_labels(cols, D) = \bigoplus_{i=1}^{m}{}_r\, h_r^{to}(cols, D_{i*})$$

$$\texttt{from\_labels} : from\_labels(D) = \bigoplus_{i=1}^{m}{}_r\, h_r^{from}(D_{i*})$$

$$\texttt{transpose} : transpose_*(D) = \bigoplus_{i=1}^{m}{}_r \bigoplus_{j=1}^{n}{}_c\, h_*^{trans}(D_{ij})$$

outputs a unit dataframe. 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.

3.2.2 Operators for manipulating metadata. We now introduce the operators for querying and manipulating metadata.

`infer_types` and `filter_by_types`: To support 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 4 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 3.2.3.

`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 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.

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

Rulebox 3: decomposition rules for relational operators

$$\texttt{mask} : mask_*(labels, D) = \bigoplus_{i=1}^{m}{}_r \bigoplus_{j=1}^{n}{}_c\, h_*^{mask}(labels, D_{ij})$$

$$: mask_r(rnSet, D) = \bigoplus_{i=1}^{m}{}_r\, \mathbb{I}[i \in rnSet] D_{i*}$$

$$\texttt{filter} : filter_r(f_r^{flt}, D) = \bigoplus_{i=1}^{m}{}_r\, f_r^{flt}(D_{i*})$$

$$\texttt{window} : window_r(f_r^{win}, w, D) = \bigoplus_{i=1}^{m}{}_r \bigoplus_{j=1}^{n}{}_c\, f_r^{win}(\bigoplus_{k=j}^{j+w}{}_c\, D_{ik})$$

$$\texttt{sort} : sort(cols, D) = \bigoplus_{i=1}^{m}{}_{o(cols)}\, h_o^{sort}(cols, D_{[p_i, p_{i+1})})$$

$$\text{where } D_{[p_i,p_{i+1})} = filter_r(p_i \leqslant cols < p_{i+1}, D)$$

$$\texttt{join} : join(cols^l, D^l, cols^r, D^r) = join(\bigoplus_g D_k^l, \bigoplus_g D_k^r)$$

$$= \bigoplus_g cross\_prod(D_k^l, D_k^r)$$

$$\text{where } D_k^l = filter_r(cols^l = k, D^l)$$

$$D_k^r = filter_r(cols^r = k, D^r)$$

$$\texttt{concat} : concat_r^{out}(D^1, D^2) = \bigoplus_{k\in\{1,2\}}{}_r \bigoplus_{i=1}^{m_k}{}_r\, h_r^{out}(labels_{out}, D_{i*}^k)$$

$$: concat_r^{in}(D^1, D^2) = \bigoplus_{k\in\{1,2\}}{}_r \bigoplus_{i=1}^{m_k}{}_r\, mask_r(labels_{in}, D_{i*}^k)$$

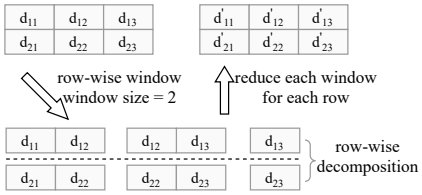


Figure 3: An example of window operator

`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 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 3 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 3). 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 $mask_r(labels_{in}, D_{i*}^k)$).

Our decomposition rules are not covered by existing systems for three reasons. First, our rules maintain logical order (i.e., append maintains the ordering). Second, we define rules for operators that are unique to dataframes, such as `to_label` and `from_label`. Third, we consider five different types of decompositions whereas relational or array-oriented databases only support a subset of the decomposition rules covered by MODIN. Specifically, relational databases do not support cell-wise and column-wise decomposition because its semantics do not support applying the same operation in parallel to each cell or column. Array-oriented databases do not support rowGroup-wise and rowOrderGroup-wise decompositions.

3.3 Applying Decomposition Rules

MODIN selects the decomposition rules based on the corresponding dataframe operations written by users. For example, the `axis` parameter in a `sum` operation determines whether one is summing over rows or columns, which subsequently determines if we should use row-wise or column-wise decomposition. After assigning a decomposition rule to each operator, we decide on the mechanisms for communicating data across different operators. If one operator outputs data to another operator and they have the same decomposition rules, then we will pipeline the output data. Otherwise, we exchange data across the two operators.

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

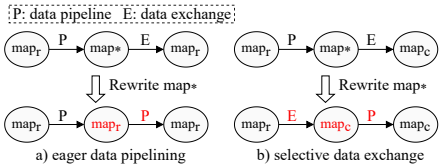


Figure 4: Optimization opportunities from applying different decomposition rules

In addition, we identify two potential optimization opportunities if we choose the decomposition rules intelligently. 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 2. Choosing different decomposition rules for the same set of operators can result in different performance. Our experiments in Section 6.4 demonstrate that selecting decomposition rules appropriately can significantly improve the performance of MODIN.

Eager data pipelining. This optimization applies the decomposition rules to allow more data pipelining. Figure 4(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 [29] 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 4(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 4(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 4(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 4(b)). This plan

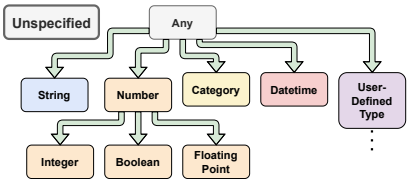


Figure 5: Dataframe Type System Hierarchy

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 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.

The two aforementioned optimizations are not possible in other systems, such as Dask Dataframe, because it only supports row-wise decomposition. MODIN supports the two optimizations due to its flexible decomposition rules. To apply the two optimizations, MODIN can use a cost model to quantify the cost of each candidate plan (e.g., choosing when to pipeline and when to exchange data) and choose one with the minimal cost. MODIN currently does not automatically support applying the two optimizations. Integrating them into a holistic optimizer is left as future work.

4 DATAFRAME METADATA MANAGEMENT

MODIN manages various types of metadata: data types, row/column labels, and mapping between the logical order of columns and rows to the physical order. We employ metadata independence, i.e., metadata is logically maintained and decoupled from its physical representation. Metadata independence enables lazy materialization to reduce overheads while ensuring correct semantics.

4.1 Data Types

Unlike relations, columns in a dataframe can have mixed types, which poses multiple unique challenges. First, querying the type of a column may require a full scan of that column. In addition, dataframe operations can change the type of a column in a data-dependent way (e.g., `map`). To maintain precise types information, we need perform column scanning repeatedly. Next, we need to formally define the semantics of a column with mixed types and how each operator modifies types information, especially when the output type is data-dependent. Finally, this type system needs to be extensible to support new types defined by users.

We propose a hierarchical type system for dataframes to address the aforementioned challenges. We define how the core operators modify types. With the clear semantics defined, MODIN can defer type inference to when it is absolutely necessary.

Dataframe Type System. Our type system supports mixed types, unspecified types, and type inference. Types are organized into a hierarchy; Figure 5 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. All types, except `ANY` and `UNSPECIFIED`, are basic types and inherit `ANY`, but MODIN can support more complicated types using the proposed type system. `NULL`s in MODIN have the same semantics as `NULL`s in relational databases. Our type system defines types only along columns and follows two invariants.

Invariant 4.1. *The output column types of the operators that accept a UDF/SDF is either provided at invocation or designated as `UNSPECIFIED` and implicitly inferred. 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 helps avoid calculating and materializing type information when it may never be needed by the user. Note 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 4.2. *A dataframe column i's type T_i is always correct, even though T_i may not be the most precise type for i.*

MODIN does not implicitly recalculate materialized types, 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. Our type system gives users the flexibility to defer type inference for performance, or to match pandas semantics that uses eager type inference by calling `infer_types` after a given pandas function.

Type Rules by Operator. Each operator from Section 3.2 has two rules for handling 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 1 describes data type handling rules for each operator. The column"Input Types" lists whether the data types must be specified before that operator is applied. For `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 1). The "Output Types" column lists 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, for a `filter`, types are not modified, even if they are `UNSPECIFIED`. For operators that allow a UDF/SDF as input, the output types can be specified by the developer (i.e., "Specified" in Table 1), or left unspecified. Suppose 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.

We believe our type system provides a consistent and correct semantics to support mixed types. For example, unlike pandas, we will not cast floating points to integers when they are mixed in a column. This maintains the accuracy of a column with such mixed types. In addition, different from pandas that does not support `NULL` in integer columns, MODIN natively supports `NULL` and produces

Table 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

correct results for left, right, and outer joins. Finally, this type system allows MODIN to easily support new user-defined types while maintaining consistent semantics.

While the high-level idea of lazy type inference has been adopted before [31], our contribution is the type system in the dataframe context and the formal description of the type semantics of dataframe operators, which enables the lazy type inference optimization.

We note that the type system does not change the physical data types, but defines the type of a column with mixed physical data types and allows for deferring the type inference. The physical data types of the data entries in a dataframe are determined by users' programs. For example, a `map` operator can output data entries with different types (e.g., int and float). Since the output data type of this operator is undetermined (i.e., `UNSPECIFIED`), we can always infer the type by physically scanning the dataframe with mixed int and float, and return a number type. Since this process is time-consuming, we defer the type inference until required.

4.2 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 vice versa (i.e., `from_label`). In addition to the flexibility of the labels, there are also latency expectations for `mask`. Thus, the system must be able to quickly execute queries on the labels, while also remaining flexible enough to move the labels into the data. We address 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 is changed and the other set is accessed. For example, `rename` can change the column labels. If it is followed by a `map` operator that adopts column-wise decomposition, we do not need to synchronize the column labels because this `map` operator does not need to access the labels. Unlike regular caching, our label caching is aware of the semantics of dataframe operators, enabling lazy label synchronization.

Logical Order management. Dataframes are logically ordered. This logical order provides a consistent view of the data: after each transformation, the rows/columns are shown in the same order. Each row/column is also associated with a numeric offset, or *position*; users can select rows/columns based on this position via `mask`.

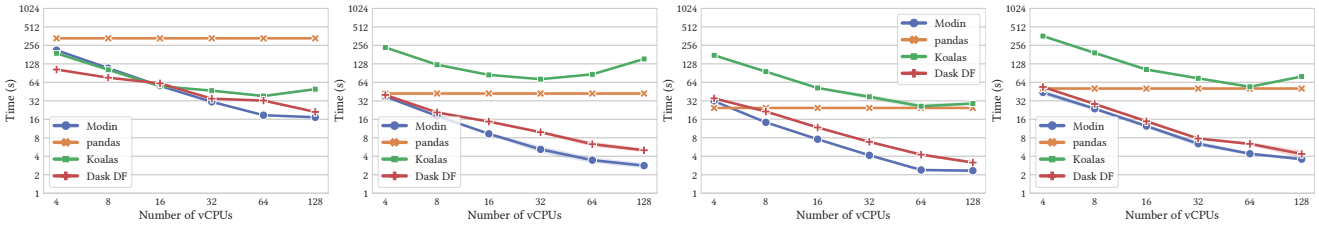
Table 2: Order and position needs and changes by operator.

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

In systems like pandas, the logical and physical layer are tightly coupled. Instead, we propose a logical order management system that maintains the logical order and physical positions separately, that is, MODIN will eagerly maintain the logical order, but lazily materialize positions, computing them when needed. This is because materializing and maintaining positions is costly, and positions are not frequently used. For example, a `filter` along rows does not change the order of the rows, but changes the positions of many rows. Maintaining these positions eagerly is costly since it requires a full scan of the dataframe.

The rules for order and position materialization and updates for each operator are listed in Table 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◇ 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.

The last column of Table 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).



(a) `read_csv` (b) `fillna` (`map` along rows) (c) `count` (`reduce`) (d) `groupby.count` (`groupby`)

Figure 6: Scalability of operators supported by MODIN and the baselines

5 PHYSICAL LAYOUT

In this section, we discuss MODIN's physical layout that flexibly supports different types of decompositions and correctly maintains the unique metadata of dataframes.

A dataframe in MODIN is physically partitioned into blocks along both the column and row axis such that MODIN can easily support row-wise, column-wise, and cell-wise decompositions without repartitioning the data. Each data block, in addition to storing the data entries, stores metadata including partial row/column labels of the dataframe, and the type and ordering information within that block. Therefore, a data block can be regarded as a "mini" dataframe and is currently implemented using the data format of pandas dataframes. To maintain global row/column order across blocks, MODIN assigns each block a number based on their order and stores the assigned numbers in a metadata manager.

An operator that uses row-wise or column-wise decomposition may require shipping data across worker nodes. Therefore, the block placement policy can impact MODIN's performance. By default, we prioritize placing blocks that belong to the same columns in the same node because column-wise access and manipulation are far more common than row-wise counterparts. Choosing the best placement policy is left for future work. For rowGroup-wise and rowOrderGroup-wise decompositions, we need to repartition the dataframe and maintain the order as discussed in Section 4.

While MODIN's physical layout design is not this paper's major contribution, it is different from existing systems, such as array-oriented databases [22, 41, 46], mainly due to the unique semantics of a dataframe. First, a dataframe needs to explicitly maintain ordering information because row/column labels do not capture it. By contrast, the ordering information of an array is stored in the attributes of the dimensions and does not need to be maintained separately. In addition, a dataframe needs to maintain the positions, which is not a concern for arrays because the dimensions of arrays are immutable when the arrays are initially created. Second, a dataframe has different types of data entries while an array adopts a uniform tuple type. The unique type semantics in dataframes enables new operations that query or manipulate data by type (e.g., `filter_by_types`). To accelerate these operations, a dataframe needs to additionally store the type information in each data block.

6 EVALUATION

Our experiments address the following questions:

- Compared to existing dataframe systems, including Koalas [4], Dask Dataframe [11], and pandas [5], how well does MODIN scale dataframe operations over a large number of CPU cores? (Section 6.2 and Section 6.3)
- How much do the optimization techniques, eager data pipelining and selective data exchange, reduce the execution time? (Section 6.4)
- What is the end-to-end performance of MODIN and how much does lazy type inference reduce the execution time? (Section 6.5)

Experiments in Sections 6.2, 6.3, and 6.4 are run on an AWS instance x1e.32xlarge with 3904 GB of memory and 128 vCores, running Ubuntu 20.04 OS. Experiments in Section 6.5 are run on an x1e.4xlarge with 16 vCores and 488 GB of memory running Ubuntu 20.04 OS. MODIN is implemented in around 62k lines of Python code and the source code is available at https://github.com/modin-project/modin. In our experiments, we choose Ray as the execution engine for MODIN.

6.1 Experiment setup

Benchmark. We use the NYC Yellow Taxi Dataset 2015 [38] with 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`. These functions cover most stages of a typical data science lifecycle, such as ingestion (e.g., `read_csv`), cleaning (e.g., `fillna`), and analysis (e.g., `join`). We additionally test two operators that manipulate the metadata: `from_labels` and `infer_types`. We also use this dataset to test the optimization opportunities when choosing the best rewriting rules from Section 3.3.

We use two additional datasets to evaluate end-to-end performance and benefits of lazy type inference. The first dataset, Loan data, [10] is from Kaggle [13] contains 2260668 rows and 145 columns, and occupies 1.2 GB of data on disk. The second dataset is the California state data from the Open Policing Dataset [15], which contains 31778515 rows and 21 columns, and occupies 231 MB of data (compressed). Though the Open Policing dataset size is roughly the same size (uncompressed) as the Loan data, it includes mostly text, allowing us to test how MODIN handles text. There is also a significant difference in the amount of skew in each of the datasets, where the Loan data is heavily skewed, and significantly more sparse, and the Open Policing data is more uniform and dense. For the Loan data, we use a notebook [9] from Kaggle [13] and deduplicate the operators of this notebook. Its key operators include `fillna`, `dropna`, `filter`, and `value_counts`. For Open Policing Dataset, we build a new notebook that follows the same workflow of the notebook for Loan data to provide a frame of reference for how data types and data skew can affect performance.

Baselines. We compare MODIN with three popular dataframe systems, pandas [5], Koalas [4], and Dask DataFrame [11] (denoted `Dask DF` in this section)—not to be confused with the Dask parallel compute framework [44]. pandas is the most popular dataframe

system in use; however, pandas runs on a single thread and does not support out-of-core computation. Koalas and Dask DF are designed to scale a subset of the pandas API and allow the working dataset to be larger than memory. Koalas translates the subset of the pandas API supported by Spark SQL (approximately 55% [42]) to leverage Spark's distributed computation framework to scale computation. Since Koalas translates to Spark SQL, it cannot support flexible operators that decompose column-wise, does not maintain the logical order, and adds additional user requirements like managing partitioning. Dask DF scales the pandas API using a light-weight native row-store implementation on Dask, a library for parallelizing Python applications. Like Koalas, Dask DF supports approximately 55% of the pandas API [42]. For example, Dask DF does not support operations that decompose column-wise, like `median` (`quantile`), `map`, and columnar `filter`s, which are critical and common to workloads that operate along columns, meaning that common feature engineering tasks are not supported in Dask DF. In addition, Dask DF does not support `iloc` (`mask` based on position) or any position-based logic, which are critical to operations that rely on the user's order (e.g., `window` functions commonly used in interpolation and data cleaning). This is due to one of the main limitations of the Dask DF data model, which forces the data to always be both logically and physically stored in the sorted order of the row labels. In practice, this means that the user cannot sort by one column and have a separate, meaningful column for the row labels. In fact, Dask DF does not support a `sort` API because of this data model limitation. Some window functions are supported in Dask DF, but they are only applied to the data in the sorted order of the index, as opposed to the user's logical order of the data. MODIN differs from Dask DF in many ways: 1) Dask DF sorts the rows based on the row labels for fast row lookups, while MODIN maintains the semantics of the logical order (as necessary for emulating pandas semantics) and builds an indexing structure on the row labels; 2) MODIN can decompose dataframes at different granularities, but Dask DF only decomposes dataframes row-wise; 3) MODIN employs its own type system while Dask DF uses the pandas type system, which is not always semantically consistent, as discussed in Section 4. We note that although MODIN and pandas have different type systems, the version of MODIN used in the experiments mimics pandas semantics to ensure a fair comparison.

6.2 Operators supported by all systems

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.

Note 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 Dask DF; 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.

[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.

Figure 6 shows the test results. MODIN has the lowest execution time compared to the baselines for all operators, because it parallelizes these operators and lazily computes metadata. pandas does not scale because it runs on a single thread. Koalas and Dask DF can scale these operators because these operators can be implemented using row-wise decomposition. Koalas has higher execution time than pandas and other systems for `fillna`, `count`, and `groupby.count` due to the overhead of Spark and an extra phase of sorting the output rows to maintain the natural order.

6.3 Operators not supported by all baselines

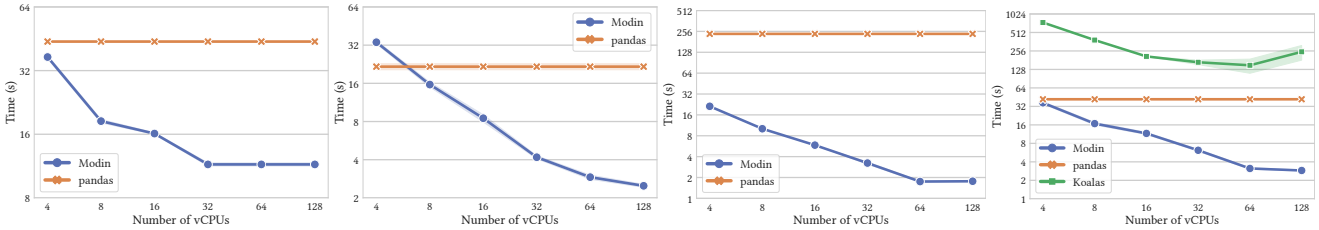
We now test operators that are not supported by all baselines, including `median`, `from_labels`, `infer_types`, and `join`. The baselines do not support these operators because they do not support operating on the column axis (e.g., computing the median for each column), the systems run out of memory (e.g., `join` for Dask DF), and 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 7a shows the scalability for this operation. 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 columns, so increasing the number of cores beyond 20 (the number of columns) does not improve the performance.

Figure 7b and Figure 7c 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 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 10× 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×.

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 7d. 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 the underlying 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 uses the same join query as in Figure 7d, varies the number of rows of the NYC dataset, and fixes the number of CPUs to 128. Figure 8 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× faster than Dask DF.



(a) `median` (b) `from_labels` (c) `infer_types` (d) `join`

Figure 7: Scalability of operators not supported by all baselines

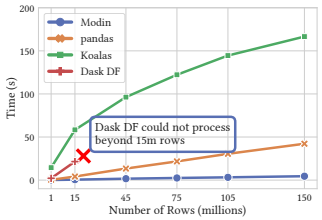


Figure 8: Join performance under varied number of rows

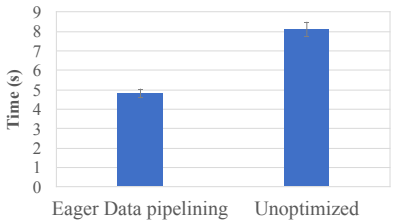


Figure 9: Performance benefit of eager data pipelining

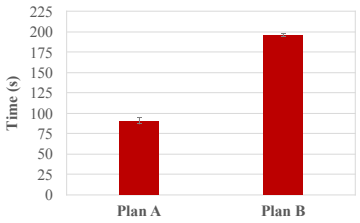


Figure 10: Performance benefit of selective data exchange

6.4 Choosing decomposition rules

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 9 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 10 shows the results of the two plans: the execution time of PlanA is 35% of PlanB. Therefore, our decomposition rules allow more optimization opportunities.

6.5 End-to-end performance

We now evaluate the end-to-end workflow performance of MODIN compared with pandas. Our comparison does not include Dask DF and Koalas because each was unable to complete the workflow, even with significant modification. Dask DF did not support `dropna` with an `axis` argument. Koalas did not support the notebook's use of `loc`. We additionally show the performance impact of our three optimization techniques: the decomposition rules from Section 3.2, the optimization involving data pipelining from Section 3.3, and the lazy type inference from Section 4.1. To do this, we test three variants of MODIN, where the first variant only includes parallelism from decomposition rules (ModinP), the second additionally includes lazy type inference (ModinPT), and the third variant includes pipelining as well (ModinPTP).

Evaluation on Loan data. The results for Loan data are shown in Figure 11a. We see that our decomposition rules can significantly reduce the execution time via parallel execution. We observe a moderate performance benefit of lazy type inference, reducing the execution time by about 0.5s. The overhead of type inference in this workflow is not large because, like many Kaggle datasets, the dataset is mostly provided clean upfront. After data types are initially inferred, subsequent operators do not need to change the data types, which does not trigger additional type inference. We test a modified notebook next to examine the overhead of type inference. Finally, we see that the pipelining can further reduce the execution time. Overall, the three techniques show that MODIN can reduce the execution time of pandas by over 8×.

Expanded evaluation on the type inference overhead. To evaluate the benefit of lazy type inference, we modify the above workflow to add three additional `map` operators that leave the data unchanged, but also do not specify output types, which will trigger type inference. This simulates a set of data cleaning operations without changing the outputs of the notebook. The results are shown in Figure 11b, we see that lazy type inference can further reduce execution time by **20%** compared to raw parallelism, despite there being only three additional cases where the types need to be inferred. In this workload,

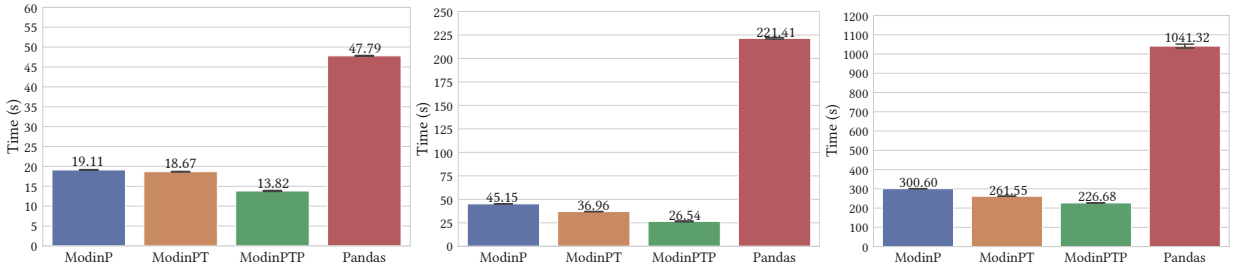


Figure 11: The end-to-end performance of MODIN and pandas on datasets: (a) loan (b) modified loan (c) open policing.

pipelining further reduces the execution time by **38%** for a total of over 8× faster than pandas.

Evaluation on the Open Policing data. Figure 11c shows the results on the Open policing dataset. Despite the workflow being identical to that in the Loan data and the datasets being similar in size, the overall runtime is almost 5× longer than the modified Loan data workflow. This runtime difference is primarily due to the differences in data types, data skew, and data density.

7 RELATED WORK

Historically, many systems 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.

Systems that support dataframe operations. Original implementations of dataframe systems include pandas [5] and R [43]. R dataframes suffer similar limitations to pandas in that they cannot exceed main-memory [7] and run on a single thread. Projects like Tidyverse [50] 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. Dask DataFrame [11] partitions a dataframe along rows to make operations along the row axis more scalable, similar to a relational database. Vaex [8] is a system for imperatively querying static memory-mapped HDF5 files, supporting around 35-40% of the functionalities of the pandas API.

There are many systems that support a subset of the pandas API via relational databases using various flavors of SQL. Koalas [4], an open-source project, translates 55% of the pandas into the API Spark SQL API via ANSII SQL. Ibis [12] translates a small subset of the pandas API into a variety of database backends Recently, there is work on choosing database backends [32] and translation into database systems like A-frame [45], Grizzly [34], and AIDA [27]. RIOT [52] achieved similar goals of employing a database backend for operating on R data beyond main-memory. Our prior vision paper introduced scalable dataframe systems and a candidate algebra [42]. We also introduced opportunistic evaluation to execute dataframe operations together in the background asynchronously [51]; this is orthogonal to the techniques proposed in this paper.

Parallel/distributed database systems. Many parallel and distributed databases [40], such as Teradata [16], HadoopDB [17], and SparkSQL [19], partition data into rows using hash or range-based partitioning to parallelize row-oriented relational operators. Additionally, column stores, like C-Store [47], Dremel [35], MonetDB [21], BigTable [24], and HBase [49], 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 BigQuery [36], RedShift [30], Synapse [18], Snowflake [26], Impala [20], MongoDB [25], 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 optimizes operators that query and update metadata, and operate along 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 [23, 39, 53], 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 [22, 48] or TileDB [41], target structured workloads, with well defined schemas optimized for scientific workloads, making them ill-suited for handling the flexible dataframe data model. We compared the physical layout in MODIN and array-oriented databases in Section 5 and discussed differences in decomposition rules in Section 3.2.

8 CONCLUSION AND FUTURE WORK

In this paper, we targeted the dual challenges of scalability and semantics underlying dataframes. We introduced 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. These techniques enable MODIN 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× relative to other partial and full dataframe implementations. In future work, we plan to extend our decomposition rules by applying full-fledged query planning and cost-based optimization across a sequence of dataframe operations. As we ponder the future of dataframe systems, we plan to continue to support, empower, and draw inspiration from MODIN's many users and contributors.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We acknowledge support from grants IIS-2129008, IIS-1940759, and IIS-1940757 awarded by the NSF, and funds from the Alfred P. Sloan Foundation. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies and organizations.

REFERENCES

[1] 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.

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

[3] What's the future of the pandas library? https://www.dataschool.io/future-of-pandas, 2018.

[4] Koalas: pandas api on apache spark. https://koalas.readthedocs.io/en/latest/, 2019.

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

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

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

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

[9] Kaggle notebook. https://www.kaggle.com/rsrinivasaraghavan/lending-club-risk-analysis-and-metrics, 2020. Date accessed: 2020-4-12.

[10] Lending club data. https://www.kaggle.com/ethon0426/lending-club-20072020q1, 2020. Date accessed: 2020-4-12.

[11] Dask dataframe api reference. https://docs.dask.org/en/latest/dataframe-api.html, 2021.

[12] Ibis documentation, 2021.

[13] Kaggle. https://kaggle.com, 2021. Date accessed: 2021-6-25.

[14] Spark dataframe api reference. https://spark.apache.org/docs/1.6.3/api/java/org/apache/spark/sql/DataFrame.html, 2021.

[15] The stanford open policing project. https://openpolicing.stanford.edu/, 2021. Date accessed: 2021-7-6.

[16] Teradata | data analytics for a hybrid multi-cloud world, 2021.

[17] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[18] J. Aguilar-Saborit and R. Ramakrishnan. POLARIS: the distributed SQL engine in azure synapse. *Proc. VLDB Endow.*, 13(12):3204–3216, 2020.

[19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[20] M. Bittorf, T. Bobrovytsky, C. Erickson, M. G. D. Hecht, M. Kuff, D. K. A. Leblang, N. Robinson, D. R. S. Rus, J. Wanderman, and M. M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th biennial conference on innovative data systems research*, 2015.

[21] P. A. Boncz, M. Zukowski, and N. 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*, pages 225–237. www.cidrdb.org, 2005.

[22] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 963–968. ACM, 2010.

[23] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on parallel and distributed systems*, 10(7):673–693, 1999.

[24] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[25] K. Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[26] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016.

[27] J. V. D'silva, F. D. Moor, and B. Kemme. Aida - abstraction for advanced in-database analytics. *PVLDB*, 11:1400–1413, 2018.

[28] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU press, 2013.

[29] G. Graefe. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[30] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1917–1923. ACM, 2015.

[31] C. Hankin and D. L. Métayer. Lazy type inference and program analysis. *Sci. Comput. Program.*, 25(2-3):219–249, 1995.

[32] A. Jindal, K. V. Emani, M. Daum, O. Poppe, B. Haynes, A. Pavlenko, A. Gupta, K. Ramachandra, C. Curino, A. Mueller, et al. Magpie: Python at speed and scale using cloud backends.

[33] S. Kläbe and S. Hagedorn. When bears get machine support: Applying machine learning models to scalable dataframes with grizzly. *Datenbanksysteme für Business, Technologie und Web (BTW 2021) 13.–17. September 2021 in Dresden, Deutschland*, page 195.

[34] S. Kläbe and S. Hagedorn. Applying machine learning models to scalable dataframes with grizzly. *BTW 2021*, 2021.

[35] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[36] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, H. Ahmadi, D. Delorey, S. Min, M. Pasumansky, and J. Shute. Dremel: A decade of interactive SQL analysis at web scale. *Proc. VLDB Endow.*, 13(12):3461–3472, 2020.

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

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

[39] R. Otazo, E. Candes, and D. K. Sodickson. Low-rank plus sparse matrix decomposition for accelerated dynamic mri with separation of background and dynamic components. *Magnetic resonance in medicine*, 73(3):1125–1136, 2015.

[40] M. T. Özsu and P. Valduriez. Distributed and parallel database systems. *ACM Computing Surveys (CSUR)*, 28(1):125–128, 1996.

[41] S. Papadopoulos, K. Datta, S. Madden, and T. G. Mattson. The tiledb array data storage manager. *Proc. VLDB Endow.*, 10(4):349–360, 2016.

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

[43] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2017.

[44] M. Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 126. Citeseer, 2015.

[45] P. Sinthong and M. J. Carey. Aframe: Extending dataframes for large-scale modern data analysis. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 359–371. IEEE, 2019.

[46] E. Soroush, M. Balazinska, and D. L. Wang. Arraystore: a storage manager for complex parallel array processing. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 253–264. ACM, 2011.

[47] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.

[48] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. Scidb: A database management system for applications with complex analytics. *Computing in Science & Engineering*, 15(3):54–62, 2013.

[49] M. N. Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605. IEEE, 2011.

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

[51] D. Xin, D. Petersohn, D. Tang, Y. Wu, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. G. Parameswaran. Enhancing the interactivity of dataframe queries by leveraging think time. In *Bulletin of the Technical Committee on Data Engineering*, volume 4. IEEE, 2021.

[52] Y. Zhang, H. Herodotou, and J. Yang. Riot: I/o-efficient numerical computing without sql. *arXiv preprint arXiv:0909.1766*, 2009.

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