Ray DataFrames: A library for parallel data analysis

Patrick Yang
Anthony D. Joseph, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-84
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-84.html

May 22, 2018

Acknowledgement

I would like to sincerely thank Professor Anthony Joseph for advising me during my can-
didacy. His advice and steadfast condence despite my uncertainty on my project were
invaluable throughout the year. I thank Professor Ion Stoica for his interest and advice on
my work.
I would also like to especially thank Devin Petersohn for his unwavering work ethic and
dedication to the Ray DataFrames project and his consistent support of my research and
goals. I thank Taner Dagdelen for introducing me to the AMP Big Data Genomics group
providing me with a research opportunity during my undergraduate career.

Finally, I would like to thank my friends and family for their support during my last four
years at Berkeley.

Ray DataFrames: A library for parallel data analysis

by

Patrick Yang

A technical report submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Anthony D. Joseph, Chair
Professor Ion Stoica

Spring 2018

Ray DataFrames: A library for parallel data analysis

Abstract

Ray DataFrames: A library for parallel data analysis

by

Patrick Yang

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Anthony D. Joseph, Chair

The combined growth of data science and big datasets has increased the performance requirements for running data analysis experiments and workflows. However, popular data science toolkits such as Pandas have not adapted to the technical demands of modern multi-core, parallel hardware. As such, data scientists aiming to work with large quantities of data find themselves either suffering from libraries that under-utilize modern hardware or being forced to use big data processing tools that do not adapt well to the interactive nature of exploratory data analyses.

In this report we present the foundations of Ray DataFrames, a library for large scale data analysis. Ray DataFrames emphasizes performant, parallel execution on big datasets previously deemed unwieldy for existing popular toolkits, all while importantly maintaining an interface and set of semantics similar to existing interactive data science tools. The experiments presented in this report demonstrate promising results towards developing a new generation of performant data science tools built for parallel and distributed modern hardware.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to sincerely thank Professor Anthony Joseph for advising me during my candidacy. His advice and steadfast confidence despite my uncertainty on my project were invaluable throughout the year. I thank Professor Ion Stoica for his interest and advice on my work.

I would also like to especially thank Devin Petersohn for his unwavering work ethic and dedication to the Ray DataFrames project and his consistent support of my research and goals. I thank Taner Dagdelen for introducing me to the AMP Big Data Genomics group providing me with a research opportunity during my undergraduate career.

Finally, I would like to thank my friends and family for their support during my last four years at Berkeley.

Chapter 1

Introduction

Data Science is one of the fastest growing public and private sectors with the advent of ubiquitous systems for computational statistics [14]. Tools such as the R programming language [23] and Pandas library [18] for Python have created accessible interfaces for manipulating data and performing exploratory data analysis (EDA) [33], the process of summarizing and understanding a dataset. These tools have enabled academic institutions and companies small and large to better extract insight from collected 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 investigate data science solutions.

Concurrently, the rate of data creation and collection has been accelerating over the past decade, and only seems to continue growing faster. This growth causes problems for traditional tools and systems, which are constrained to single-node, single-threaded implementations. This growth has spurred a set of responses in the distributed and parallel computation scope, with many computation frameworks [10, 4, 32, 20] embracing distributed computing across cluster systems for big data problems.

Most data scientists, who may have trained backgrounds in mathematics or statistics instead of computer science, are not trained systems engineers. As such, they may have the knowledge to manipulate or infer conclusions about data, but may be unfamiliar with best practices for engineering fast, scalable software to work with said data. Clearly it is unreasonable for a data scientist to work directly with low level libraries such as MPI, and even interfaces that computer scientists label as high level, such as MapReduce or Spark RDDs may be out of reach for a typical data scientist. The average data scientist would gladly prefer an abstraction such as Pandas, R, or Spark SQL to work with their data.

On the other hand, data scientists typically interact with their data in interactive environments, using ad-hoc methods. For example, in Python, a typical data science workflow may start by initially inspecting the collection or store of data by loading it into Pandas, performing a variety of cleaning methods based on their manual inspection of the data, and displaying summary statistics on the data, all in an interactive environment such as IPython [21] or Jupyter Notebook [15]. The actions they 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 very disjoint from the view provided by many existing popular parallel computation libraries, such as Dask [26], Spark [4], and Tensorflow [1], which design computation around building and lazily executing entire dataflow graphs. The advantage to the aforementioned designs is the ability to perform query optimization on the entire query, and to better schedule individual portions of the query job, compared to eagerly evaluated frameworks. However, data scientists are accustomed to eager evaluation-based tools, which have more natural interoperability with interactive environments.

As such, to better enable data science in the wake of big data and parallel computations, we propose Ray DataFrames, an evolution of existing DataFrame libraries. We envision Ray DataFrames as a successor to libraries such as Pandas, with the capability of providing scalable, performant tools on parallel systems while maintaining familiar semantics and an equivalent user-level interface.

Chapter 2

Motivations

The initial goal for Ray DataFrames was to allow existing Python users engaging with exploratory data analysis on Pandas the capability to scale their analyses from small datasets on the order of kilobytes, to massive datasets on the order of gigabytes. One key motivating factor behind the design was to ensure that users would not only be able quickly migrate to Ray DataFrames, but also that any existing code, scripts, or notebooks would stay static regardless of the scale of data used. One could imagine this being very useful, for example, if a user would prefer to test hypotheses or workflows on small data for rapid iteration, and then use the same code for an entire massive dataset without modification.

To achieve that goal, we export the same API as Pandas, which allows existing Pandas scripts or notebooks immediate interoperability with Ray DataFrames through a single import change[1]. Users who are already familiar with the Pandas interface will immediately know how to use Ray DataFrames, as we engineered our API for concordance and feature parity with the existing Pandas API.

Apart from our user level API, to explain our engineering motivations in Ray DataFrames, we first consider a typical EDA workflow to illustrate both our design focuses and design choices:

Exploratory data analysis first begins with **importing data**, which involves reading data from an external source or store. Typically, these sources will be a text file, the most popular format being CSV or other related delimited formats, and other forms of tabular and columnar files. The user will then proceed to **data wrangling**, which typically involves examining the types of data imported, finding the amount of data, and printing out the first or last few rows of data, to provide a feel of the data. This step may also involve data cleaning or enacting other small mutations to the dataset in order to coerce the dataset into a usable form. After the data is successfully explored and cleaned, the user proceeds to **exploratory data modeling**, which involves operations performed across the dataset and deriving new datasets from the existing dataset to build insights from the original data. The user will then

[1]Currently, this is only true for scripts and notebooks with direct imports. Libraries with Pandas as a dependency (such as `xarray`) would require custom versions of those libraries to use Ray DataFrames, and those dependent libraries are currently unsupported

perform **data visualization and communication**, which involves generating charts and graphs, summarizing the insights gained from the experiment, or packaging the generated model into a usable form for later. Lastly, the user may **export results** of the experiment, which usually involves writing derived or cleaned datasets or models to an external store.

This process helps us organize our requirements for Ray DataFrames into three categories.

Interactivity

The key realization for designing systems for EDA is that a typical EDA workflow is fundamentally an iterative process, as is the case with many data science workflows. In other words, the execution of the steps taken, or even the existence of steps themselves, can be changed on a whim based on the results from a previous step in the experiment. Likewise, the structure of said steps is fluid in practice; a data scientist may choose to explore different forms of manipulating or wrangling the dataset after receiving poor experimental results, or may choose to import more data to augment the experiment. In this way, the process taken for data science is distinct enough from engineering that it could be classified more closely to traditional science fields, although engineering and data science are implied to be closely related.

It should be clear why data scientists favor tools that provide high degrees of interactivity, such as shells for Python or R or notebooks. Compare this paradigm to that presented by systems such as Dask, Tensorflow, or Spark, where actions are first declared to generate a dataflow graph, then executed only when the user explicitly initiates the graph computation. In the case of Dask or Tensorflow, explicit commands (`df.compute` in Dask, `session.run` in Tensorflow) are required to execute computations on the graph, while in Spark, computation only occurs when an "action" (a command generating output). These artifacts of lazy evaluation do not synchronize with interactive interfaces, since interactive users generally have the expectation cognitively of gradually constructing the end result solution. Likewise, debugging lazy evaluation systems is much more difficult than traditional eager systems, due to the inability to step through computation steps at a fine grain or the ability to inspect intermediary points of computation.

Ray DataFrames exposes a DataFrame API that acts exactly like dataframe in eager evaluation interfaces such as R or Pandas. The user can execute individual operations on a dataframe and use output functions to gradually verify the data within a dataframe. Computation tasks within a dataframe start immediately upon the function being called, instead of only when output is required. For an interactive user, who may be gradually composing their set of functions to perform on a dataframe across multiple shell prompts or notebook blocks, this may lower the total computation time for the end result, as computation would be happening in the background for previous tasks while the user is still contemplating future functions to execute.

Concordance

The existing community surrounding Pandas is very large, spanning 5 million questions by 1 million unique users on Stack Overflow in October 2017 [16], and is cited as one of the largest growth factors for Python usage on Stack Overflow [25]. Of particular note is that a plurality of such additional users come from academia, who most likely use it for general purpose analysis as opposed to engineering, akin to the current role of R in academia. As such, we want to provide an interface that extends beyond familiar, making the Ray DataFrames API as concordant with the existing Pandas API as possible. We strive to make it as simple as possible for a Pandas user to test out or convert their existing scripts and notebooks to Ray DataFrames, by only requiring a single import line change. This decreases barrier to entry for new Ray DataFrames users, and also paves a road map to add support for other libraries that are frequently used with Pandas, such as scikit-learn or matplotlib, in the future.

Performance

With dataset sizes growing regularly into the tens of gigabytes, even small-scale samples of working datasets may appear in the hundreds of megabytes. While popular libraries such as Pandas, which provides good interactivity, work well on datasets in the tens of megabytes, their single-core implementations severely limit their performance when datasets grow into the multi-gigabyte range. By leveraging parallelism at the node and cluster level, we aim for Ray DataFrames to enable exploratory analyses on large datasets with significant speedups in loading and processing time. Most importantly, we want to greatly increase performance while still also supporting the other points made above.

Chapter 3

Technical Foundation

3.1 User Interface

As mentioned above, the Ray DataFrames interface emulates the existing Pandas interface. The user **simply replaces their import statement**, `import pandas as pd` with `import ray.dataframe as pd`, and uses a new library with the same functional semantics as Pandas. Ray DataFrames shadows existing Pandas class and top level functions, so under the hood `pandas.DataFrame` will turn into `ray.dataframe.DataFrame`, `pandas.concat` will become `ray.dataframe.concat`, and so on. Importantly, the user should be oblivious to this change: for an existing Pandas user with an existing script or notebook, **replacing the import statement will have the exact same functionality**.

For items that we feel are either inappropriate or inefficient to re-engineer for parallelism, such as Pandas data types and Pandas indexes, Ray DataFrames transfers imports of such items, so `ray.dataframe.Index` will simply be an alias for `pandas.Index`. This way, we cover all Pandas use cases a user might want to use, whether we decide to explicitly re-engineer it or not.

Noticeably absent from the framework is the API for specifying the number of cores in a system or cluster or data partitioning and distribution strategies. This is intentional, as Ray DataFrames seeks to transparently distribute and compute user datasets in parallel. Users should not have to worry about tuning the library for their hardware, as Ray DataFrames will perform optimization and parallel computation under the hood. Ideally, to the user, performance should be an afterthought and functionality should be their primary concern.

3.2 Execution Framework

On each parallel executor, Ray DataFrames uses Pandas as its computation engine, specifically for each per-core DataFrame partition. Pandas is also used to ensure that all com-

ponents, such as indexing or axis specification, remain fully concordant with themselves, without having to reimplement said functionality.

Ray DataFrames is a subproject under the Ray project [19, 24], and thus we leverage Ray as our underlying execution framework. Ray is a high-performance distributed execution framework targeted at large-scale machine learning and reinforcement learning applications. It achieves scalability and fault tolerance by abstracting the control state of the system in a global control store and keeping all other components stateless. Ray uses Apache Arrow [3], a modern columnar data serialization format and library specialized for in-memory data, and also uses Plasma In-Memory Object Store [22], a shared-memory distributed object store for use with Arrow. Rays close integration with Apache Arrow allows for fast and efficient serialization of existing Pandas objects, greatly improving performance for Ray DataFrames. Plasma Store's ability to support zero-copy data exchange across interprocess communication also helps improve performance for tasks on remote partitions. Ray's usage of ObjectIDs to represent pieces of data in the object store allows Ray DataFrames to maintain remote handles to data partitions and communicate said partitions across separate processes without explicitly transferring data over an RPC. Lastly, Ray has the ability to scale to hundreds or thousands of nodes in a single cluster, enabling it to operate on massive datasets.

We currently are not focused on cluster performance, and do not investigate cluster performance for Ray DataFrames in this paper. Significantly, Pandas users have remained Pandas users specifically because they care mostly about single-node performance. Therefore, we instead are currently investigating optimizing single-machine, multi-core performance, an area which Ray DataFrames still demonstrates large speedups.

Chapter 4

Technical Architecture

4.1 Partitioning

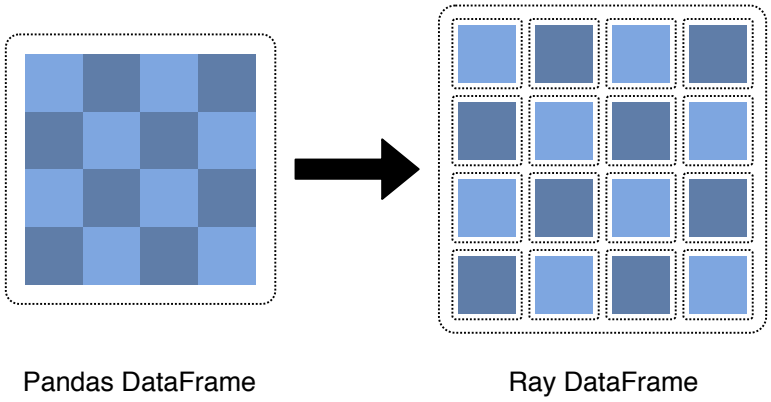

Figure 4.1: Converting Pandas to Ray DataFrames. Each of Ray DataFrames' partitions represents a certain sector within the original Pandas dataframe, which we call a *block partition*.

Each Ray DataFrame's dataframe is partitioned into block partitions, each representing a certain row and column range within the entire dataset, similar to the concept of the block matrix [36] in mathematics. Each block partition is a Pandas dataframe that has been serialized into the object store to enable remote tasks on top of the block. Block partitions are the building blocks for dataframes, and each partition's remote ObjectID is kept in the dataframe shell class on the local driver, organized into a two-dimensional NumPy array. This grid-like organization allows for easy access to individual partitions, or all the partitions across a certain axis, as well as the ability to quickly perform transpose operations.

Some dataframe operations can be described as axis-invariant, in which actions are performed generally on individual cells within a dataframe, and do not produce axis-dependent results. Many dataframe operations are axis-variant, in which they take an axis as a function parameter and perform the given action across or against the provided axis. Block partitions

allow us to support this by localizing the computation by coalescing the partitions into entire rows or columns on the fly (See Fig. 4.5).

4.2 Index Metadata

Indexes are a critical component for Pandas operations and dataframe organization, as functions such as `loc`, `iloc`, `insert`, and other label-based functions vary their behavior based on a dataframe's indexes. However, partitioning the index and maintaining it on the remote partitions is costly for performance and memory. Label-based lookups may require a traversal of multiple partitions to find a single column, and replicating the indexes on each partition duplicates data. Because it is critical to maintain one single non-partitioned view of the index in the driver memory, we have created a separate encapsulating class for a dataframe's index and column names, as well as other axis-dependent information, such as columnar dtype (data type) information. The Index Metadata object, as we refer to it, generates a mapping between an index label and a pair of coordinates, which identify the location of the label in the partition scheme. For example, for the row index, a function may lookup the coordinates for a particular row label, use the first coordinate to find the row partition number of the label, and use the second coordinate to find said row number within said partition.

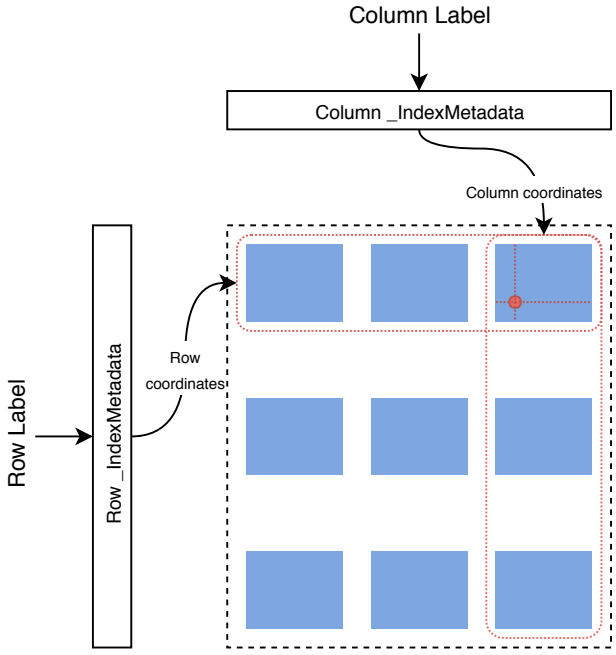

Figure 4.2: Index Metadata. Given a label on one axis, the metadata looks up the coordinates for that label. Given a row and column label, the location of that element can be precisely determined.

The metadata object is backed by a Pandas object acting as a lookup table, referred to as the *coordinate dataframe*, which maintains the actual dataframe's index, with two columns for each coordinate point. We chose this architecture in order to simplify concordance with Pandas; this way, given a user-provided index, the metadata object can immediately signify whether that provided index is valid by attempting to apply it onto the coordinate dataframe. In this way, we treat the coordinate dataframe as a sort of in-memory canary request, to save the expense of distributing the index out to the partitions, only to have it fail later.

The contents of an Index Metadata object are also generated remotely, and only collected into the driver memory when a certain operation requires either an index lookup or an index modification. This allows the dataframe to pass unmodified Index Metadata objects when performing an operation that does not mutate the index, such as `apply` or `applymap`, and also preserves the non-blocking nature of Ray DataFrames functions.

4.3 Execution Layer

Most functions within Ray DataFrames can be categorized into a certain category due to their similar natures of operation. In turn, we develop very similar processes for categories of functions, of which some are listed below.

Ingestion

Data ingestion into Ray DataFrames is done similarly to Pandas, and can be performed by turning existing Python objects into dataframes, or importing from external files. Coercing existing Python objects, such as lists, dictionaries, and NumPy arrays, currently uses the Pandas dataframe constructor to turn said object into a Pandas dataframe, which we refer to as a *row partition*. Each row partition is then further partitioned column-wise into block partitions. For file import operations, each file is first logically partitioned based on the

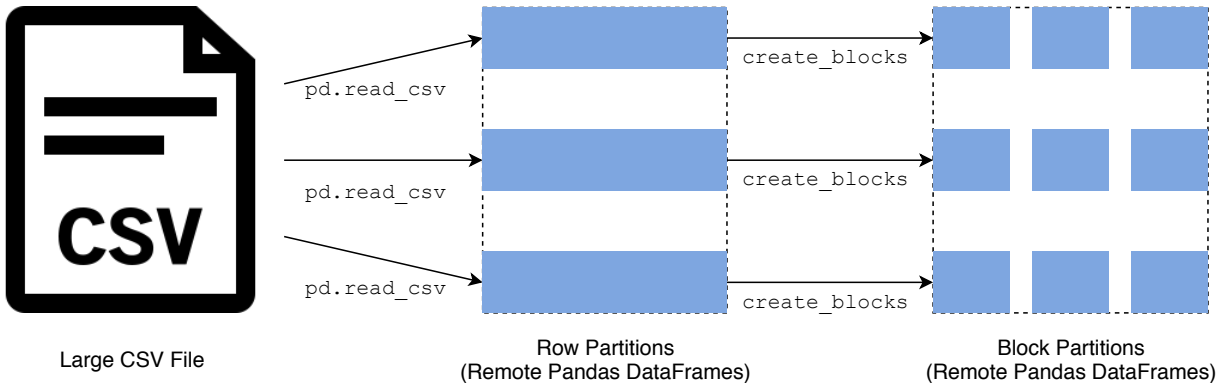

Figure 4.3: CSV Ingestion Process. The CSV is broken into different byte ranges, each of which is read with `pandas.read_csv` in parallel, remotely. Those *row partitions* are then partitioned further into *block partitions*.

size of the file in bytes, and Pandas import functions are used to turn those logical partitions into block partitions remotely. For example, for `ray.dataframe.read_csv`, the file is initially marked for partitioning by line number and separate remote tasks perform `pandas.read_csv` on those logical partitions. This generates a set of disjoint row partitions for the dataframe, and the row partitions are further converted into block partitions. This allows Ray DataFrame to ingest data from an external source in parallel, performing reads much faster.

Applymap Functions

Applymap functions in Pandas are actions on a dataframe that execute a function or lambda on each element of the dataframe distinctly, returning a dataframe of the same dimensions with modified elements. Functions such as `applymap`, `isna`, or scalar arithmetic operations on a dataframe fall into this category. Due to the axis-invariant nature of the operation,

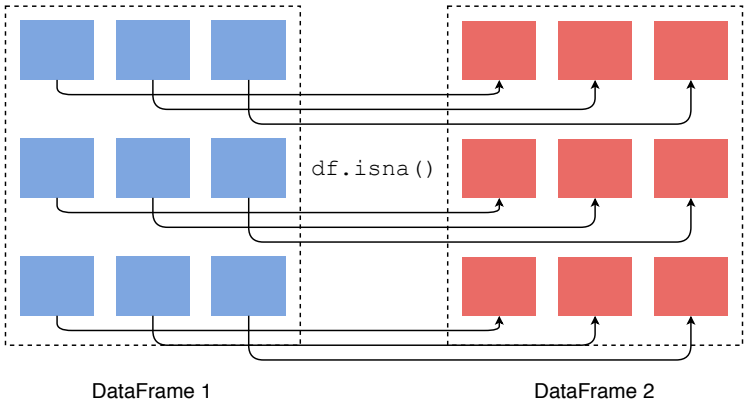

Figure 4.4: Calling `df.isna`. A remote task is invoked on each remote partition, resulting in a new set of block partitions, packaged into a new dataframe.

operations are broadcast to each of the individual block partitions, returning another grid of block partitions that we package into a new Ray DataFrame. Notably, the dataframe dimensions do not change, nor do the indexes, so those dimensionality of the block partition grid, as well as the index metadata objects, are preserved in the newly created dataframe, saving some computation time.

Axis-Variant Functions

Axis-variant functions in Pandas are actions that operate on entire rows or columns of a dataframe, as a series, and return a result either as a series or single item. Functions such as `sum`, `mean`, and `describe` fall into this category. To make the execution of such functions simple, intuitive, and robust, the block partitions are first combined into entire rows or columns, dependent on the axis of operation. The action is then instead

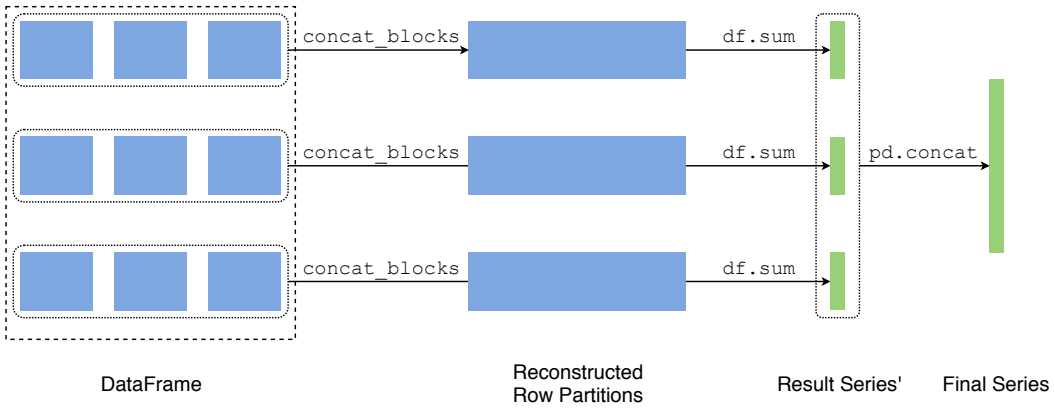

Figure 4.5: Calling `df.sum`. The block partitions are first concatenated into axis-long partitions (in this case, *row partitions*). The operation is then performed on those partitions remotely, the results collected, and put together.

invoked as a remote task onto each of the row or column partitions. This simplifies the emulation of those functions on partitions while maintaining the parallel execution across distinct partitions. For functions that reduce a series to a single scalar value, the output is returned as a concatenation of the resulting series of values from each row/column partition. Otherwise, for functions that return a complete partition, a new dataframe is built from the concatenation of the partitions.

Mutation Functions

Mutation functions in Pandas are actions that change values within a dataframe at a certain location within the indexes. Functions such as `insert`, `remove`, and the `__delitem__` override fall into this category. Execution of these functions is generally tricky, since mutations require an inplace modification of a subset of block partitions, and due to the immutable nature of Ray's object store, these partitions must replaced by completely new partitions. Initially, the index metadata objects are consulted to find the coordinates for modification in both axes, if necessary. The requisite blocks are also combined into row of column partitions, if the action involves a series. Using the coordinates, a function is then dispatched to the correct partitions to insert, remove, or modify the values specified, generating new partitions, and new partition ObjectIDs. Those new ObjectIDs are used to replace the existing stale partitions in the dataframe.

Grouping Functions

Grouping functions in Pandas are the `groupby` function and all associated actions performed on grouped data. Since the dataframe is logically partitioned across both axes, and the partitions currently do not follow any logical partitions with respect to the axis indexes or the underlying data, grouping can only be done reliably by grouping across entire column

partitions. Thus, the block partitions are first combined into column partitions and the specified grouping data is broadcast to each column partition. Since the grouping data is grouped in a deterministic fashion, performing individual remote groupings on each column partition should yield partition groups that can be reliably combined as well. All grouping operations are wrapped in a local driver object to manage any functions performed on the partition groupings, emulating the Pandas `DataFrameGroupBy` object.

Join Functions

Join functions in Pandas encompass all actions that require some sort of index coercion between two differently indexed dataframes. This includes `join`, `concat`, and `merge`, but also covers rudimentary functionality such as `add` and other inter-dataframe math operations. Since these functions implicitly perform an outer join on both dataframes passed in, they share much of the same logic with the concatenation and joining functions. The join is initially performed on the indexes for the two dataframes, and then both dataframes are reindexed into the new joined indexes. This is performed to ensure that the two dataframes are put into a co-partitioned state, such that the labels across their rows match. Depending on the function called, the resulting dataframes are then either concatenated or combined elementwise.

Export Functions

Export functions in Pandas are actions that take data stored in a dataframe and export them in some usable representation, whether that is to the user console, to a different Python object, or serialized to disk. Currently, while some functions, such as the `__repr__` override, have subtle optimizations to reduce the amount of data covered, most export functions are handled through Pandas. For example, `to_csv` performs a `to_csv` on each row partition in serial order to a single file. There is room to optimize this in the future, as this was not our primary focus on the initial release of Ray DataFrames.

4.4 Implementation Scope

Experienced Pandas users may note that the current enumeration of functionality does not directly match with the full range of functionality exported by Pandas. Some notable absences include a distributed Series implementation, Multilevel index support, and direct support for plotting. Likewise, users familiar with execution internals of other distributed dataframe libraries, such as Spark or Dask, may dispute our approach towards operations such as `df.sum` or `df.max`, in which we perform operations on concatenated block partitions as opposed to performing a two-part reduce or tree reduce. As Ray DataFrames is a living project, our currently exported functionality, to be released with Ray 0.5.0, aims to best build concordance with the existing Pandas dataframe API so prospective users can

immediately familiarize themselves with Ray DataFrames' performance. In the future, we plan to extensively expand the scope of our implementation, as well as approach other key optimizations and opportunities, such as multi-node distributed operation.

Chapter 5

Performance Optimization

Distributed and parallel computation programs always require more complex performance optimizations than serial programs, generally since the same assumptions cannot be made between a serial program and its distributed counterpart. For example, while Pandas can rely on simply enqueuing operations serially over a dataframe, more consideration must be taken for operations using multiple processors or cores across multiple partitions. These considerations include inter-process communication (IPC) and synchronization latency across different worker processes, serialization and deserialization bandwidth, and job execution time variance. For all the mentioned problems above, using partitions distributed across multiple cores presents a challenge over optimizing runtime for the entire dataframe.

5.1 Partitioning Effects

The most direct and critical optimization at the dataframe execution layer is on partitioning, specifically the size, quantity, and dimensions of block and axis partitions for which jobs are run. The partitioning scheme for a dataframe must take into account the above performance anomalies and balance any possible optimizations to partitioning to possibly reduce overhead of re-partitioning. For example, one important decision to make would be ensuring that the number of partitions is large enough to fully saturate all cores, but also small enough so as not to incur a large penalty for IPC or remote task overhead.

Further complicating the problem is the possible diversity of hardware types and configurations in which users will be performing dataframe analysis. For a single system, it would be plausible for a single experienced developer to run performance experiments to develop an optimal partitioning scheme. However, given our goal to make the parallelism within Ray DataFrames opaque, complicated by the likely inexperience of our primary userbase, it is important for the system itself to adjust to the users environment and needs automatically. The user environment may vary in properties that are easy to measure, such as number of cores or amount of available memory, or those that are harder to quantify, such as memory, disk, and serialization speed, and IPC latency.

Although machine-wide optimizations may be made on a certain machine-level granularity, such schemes may still be too broad for optimum or best-case performance, as the workload profile for each machine may change across different runs, or even within each run. As described above, each function performed on a dataframe can be binned into a certain performance archetype. These archetypes have various different optimal partition parameters, and fluidly and quietly changing partition schemes across functions would improve runtime for each function, and runtime overall. This is further complicated by Ray DataFrames eager evaluation approach to function dispatch. In a lazy evaluation system, the query optimizer can take a global approach for a certain query and engineer the best scheme for the partitioning for each function transition. However, our eager evaluation approach presents a challenge for not only determining the best partition scheme for a newly dispatched function on the fly, but also estimating the best-fit scheme for the next possible function call.

5.2 Partition Modeling

In order to best construct a partitioning scheme on-the-fly with each function, we propose an optimizer to **model and select partitioning splits by minimizing predicted runtime**. This optimizer would act as a prologue and epilogue to any function call. Our model must have the following properties:

Speed

Any model used for estimate partitioning performance for optimization purposes must be fast. Based on the timescales[1] for which functions would operate on, if the estimation model were not magnitudes faster than the execution, it would be more effective to use a naive, universal partitioning scheme, such as simply always partitioning to the number of processor cores in the system. As such, our partitioning model cannot include live runtime estimations, such as canary requests on individual partitions, simply due to the fact that such runs' times would be on the same scale as the actual function call itself. If the actual function dispatch is dependent on such an estimation, the total runtime with canary estimation would dwarf the runtime with optimal partitions. Therefore, our model must solely rely on runtime estimation over small metadata, without live testing. Previous related work has demonstrated the capability for static performance estimation with regards to data partitioning [5], providing a quick and relatively accurate gauge of application runtime.

Opacity

As mentioned in the previous section, any partitioning scheme optimizations must be made opaque to the user. We compare this to other frameworks which require the user to tune

[1] `df.isna`, for example, runs under 0.5 seconds on 2 GiB, even for poorly partitioned datasets

partitioning for optimal performance, such as Spark [29] or Dask [9]. Although expert users may have the ability to single-handedly fine-tune the performance of a workload or set of workloads on a particular machine, it is inefficient and sometimes infeasible to expect even expert users to develop and fine tune a scheme for every possible situation. Furthermore, given our expected userbase and the type of interface we export, it would be burdensome and contradictory to solely expose partition optimization options to the user.

Robustness

While performing actions on partitions, there exists a natural stochasticity for each partition's runtime, which could be based on a variety of factors with respect to the system as a whole. In general, this job performance variance is relatively unpredictable and uncontrollable with respect to estimating a certain jobs absolute runtime. For a single-executor system, this would not be an issue, since variance would be averaged out through the queue of jobs on the single-executor. However, due to the bulk synchronous parallel nature of dataframe operations, job variance can cause straggler jobs to block the progression of any future functions from beginning [2]. In other words, the overall runtime would not be the average of a single job execution, but instead be the maximum job duration. Although this phenomenon remains persistent and is more endemic in truly distributed domains, our partitioning model should remain robust despite the performance hits from stochastic job runtimes.

5.3 Function Execution Cycle

User interaction with a dataframe is represented through an execution cycle, in which each instance within the cycle is a separate call by the user on a function for the dataframe. Each instance can be broken down into a set of stages, which can be represented as functions, given an input of partition parameters and amount of data, estimate the runtime for that stage. The major stages we model are the **load** stage, the **explode and condense** stage, and the **action** stage. These stages, along with their interactions with the optimizer, are shown in Fig. 5.1. In general, the stages can be broken down as

$$F(x, y, d) = f_{op}(x, y, d) + S(x, y, d) + \epsilon(x, y, d) + \mathrm{Var}(x, y, d)$$

Where f_{op} is the **operation time** of the actual emulated function, S is the **serialization time** of the function's result, ϵ is the **task overhead** associated with the number of partitions, and Var is a function estimating **worst-case straggler latency** from tasks with maximum deviance. This proposed execution cycle can be further augmented to fit different function archetypes as well as other external parameters, as estimation for new functions would be as simple as determining a functional representation of the new function and providing a runtime estimator for the new function. We purposefully omit deserialization time in this formulation since it is partially encompassed within the serialization time cost already.

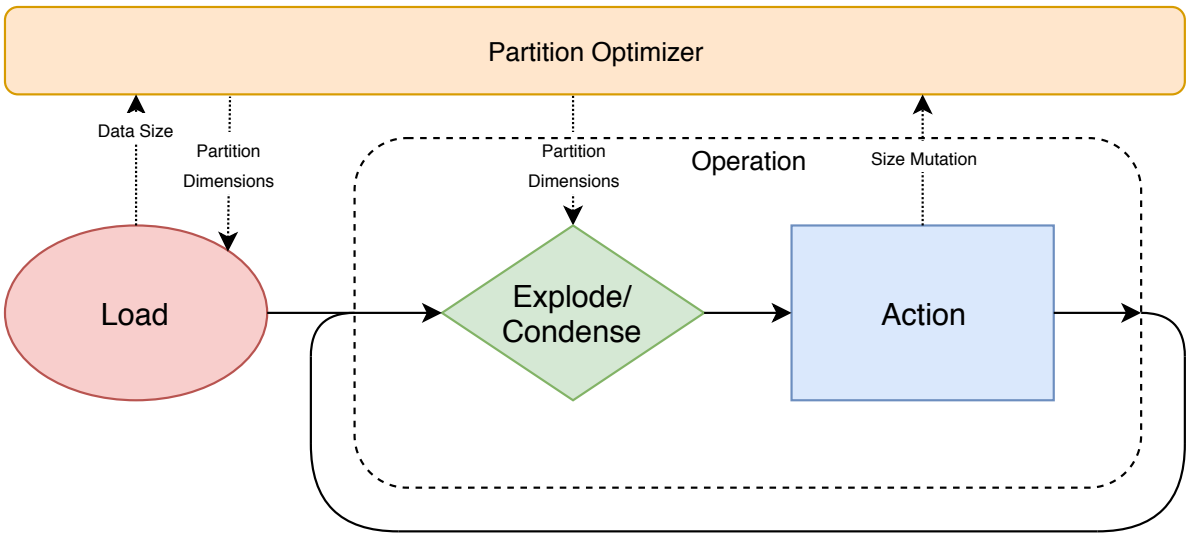

Figure 5.1: Dataframe Workflow Model. The user first loads data from disk, and then continuously performs actions on the dataframe to change the state or demonstrate results. Within this process, we add the partition optimizer and the explode and condense stage.

Likewise, Ray's usage of Apache Arrow [3] for serialization, memory layout, and its Plasma shared memory object store mean that deserialization of tensor-like objects on remote tasks is roughly constant time and insignificant enough to disregard.

Load

The load stage represents any operation that ingests data, whether from a file or a separate Python object, into a Ray DataFrame format. Performance of the load stage is modeled as

$$L(x, d) = L_D(x, d) + S(x, 1, d) + \epsilon(x, 1, d) + \text{Var}(x, 1, d)$$

where L_D represents the loading speed from disk if the load operation reads from a file. Since primary development of features in Ray DataFrames has favored row-based storage, such as CSV or plaintext files, the speedup from parallelism can be derived as a function over the number of specified row partitions.

Explode and Condense

The explode and condense stage is a bridge stage used to enable re-partitioning on-the-fly to optimize runtime. It takes in a grid of block partitions and returns a new grid of block partitions, in which the size of the grid is stretched or condensed by a certain factor in one or both dimensions. Performance of the explode stage is modeled as

$$E(x_i, y_i, x_o, y_o, d) = C(x_i, y_i, x_o, y_o, d) + \epsilon(x_o, y_o, d) + \text{Var}_E(x_o, y_o, d)$$

where the parameters x and y here represent the eventual exploded partition numbers. We add another functional effector, C, representing the time required to concatenate separate remote partitions. For this stage, the serialization penalty is not present since we treat any additional latency solely as a result of the exploding and condensing that may occur to coerce the partitions into the right shape.

Action

The action stage represents any operation performed on a dataframe to transform or output data. Performance of the action stage is modeled as

$$G(x, y, d) = f_{op}(x, y, d) + S(x, y, d_o) + \epsilon(x, y, d) + \mathrm{Var}_{op}(x, y, d)$$

where f is the action-specific runtime function, which differs depending on the Pandas native function called, and which may have a different profile dependent on the function's archetype as defined above. For example, axis-reducing functions like `df.sum` concatenate the block partitions within one row or column first, then perform the action over that concatenated axis. Broadly, we can emulate it as a function over the number of partitions in both axes and the amount of data within those partitions.

Cycle Optimization

With each of these stages, we model the entire execution of a workflow as

$$T(d) = L(x_0, y_0, d_0) + \sum_{i=1}^{N} (E(x_i, y_i, d_i) + G(x_i, y_i, d_i))$$

Given the set and ordering of functions running on a workload, optimizing the partition numbering to minimize runtime would involve finding the optimum x_i and y_i per stage to minimize the goal $T(d)$. However, due to the interactive nature of the library and our goal of eager evaluation, it is unfeasible to globally optimize for the best partition numbers prior to the user specifying the desired functions to run. Instead, we choose to locally optimize runtime greedily at each stage, not only to simplify the formulation of the final model, but also due to the explode operation providing a relatively low latency fix if more parallelism is desired. As such, for our model, given an existing state (x_{k-1}, y_{k-1}), we aim to minimize

$$T_k(d) = E(x_{k-1}, y_{k-1}, x_k, y_k, d_k) + G(x_k, y_k, d_k)$$

The main optimization will come from balancing the latency overhead from additional partitions versus the possible overhead from variance given coarse straggler jobs. In order for accurately model the total runtime latency for a given partition scheme, performance testing was done for each stage and stage component, examining experimental results of runtime given a search across possible partition numbers. These numbers were used in turn to build closed form formulas for each stage component.

5.4 Performance Model Fitting

With the above formulation, we used experimental results to attempt to model the various functions that dictated runtime of each stage. For each different function, we constructed a script with a dummy dataset in order to gather empirical data on each function's growth behavior. As demonstrated on the graphs, it was difficult to predict a priori the scaling of the various functions. For example, the condensing routine resulted in a runtime that grew experimentally at a complexity of $O(\sqrt{n}\log(n))$, as shown in Fig. 5.2. This is likely due to the time complexity of performing grouped concatenations of Pandas dataframes on a single axis. Compare this result to a relatively tame function, such as the condensing operation, which had linear complexity, as seen in Fig. 5.3.

We also present the scaling characteristics of various functions used to test the prediction modeling in Fig. 5.4, fitting in the place of the various f_{op} functions within the action stage. Fortuitously, most functions, excepting a niche operators, examined scaled linearly, allowing us to simplify the prediction modeling. However, our prediction mechanism should be extensible to functions with various sorts of runtime complexities.

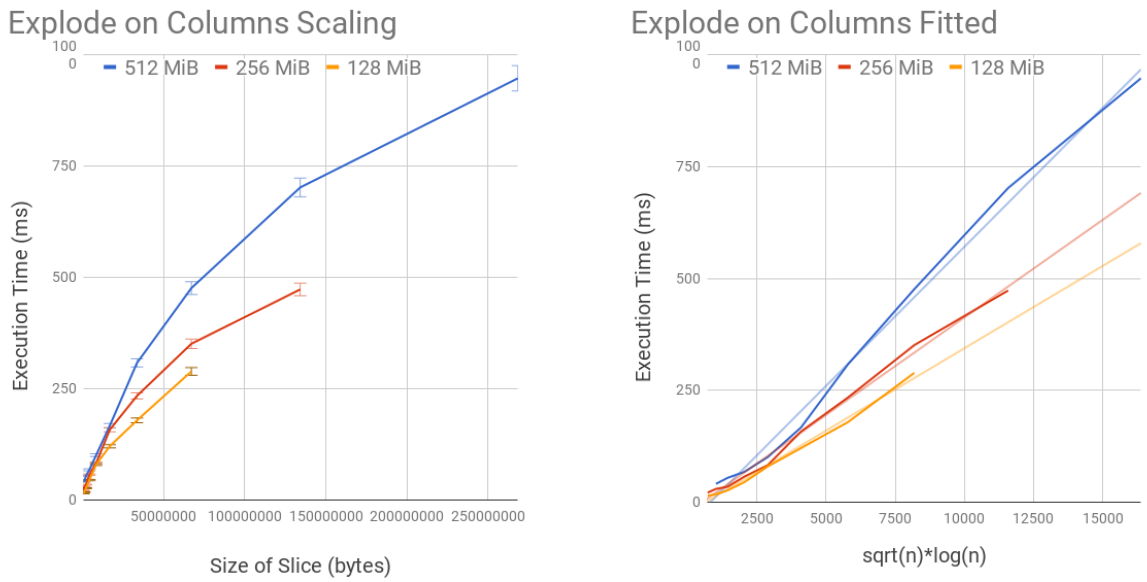

Figure 5.2: Explode operation on columns scaling. On the left, the raw results of slicing out columns of various sizes from three differently sized partitions. On the right, the horizontal azis (column slice size) has been scaled to fit $O(\sqrt{n}\log(n))$. These operations were tested across 5 runs each.

.

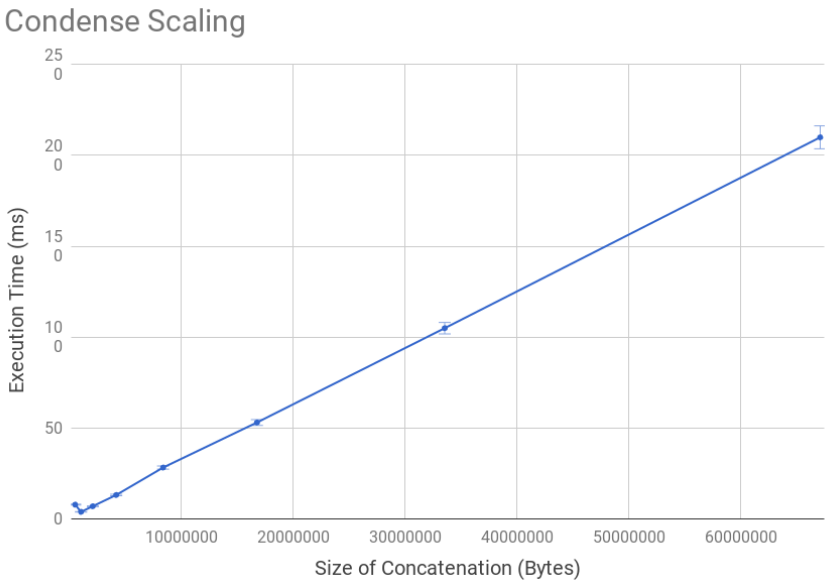

Figure 5.3: Condense operation scaling versus number of bytes to concatenate. This result was confirmed using various different sizes of partitions, with minimal variance between the runtimes on the varying basis partitions used to concatenate.These operations were tested across 5 runs each.

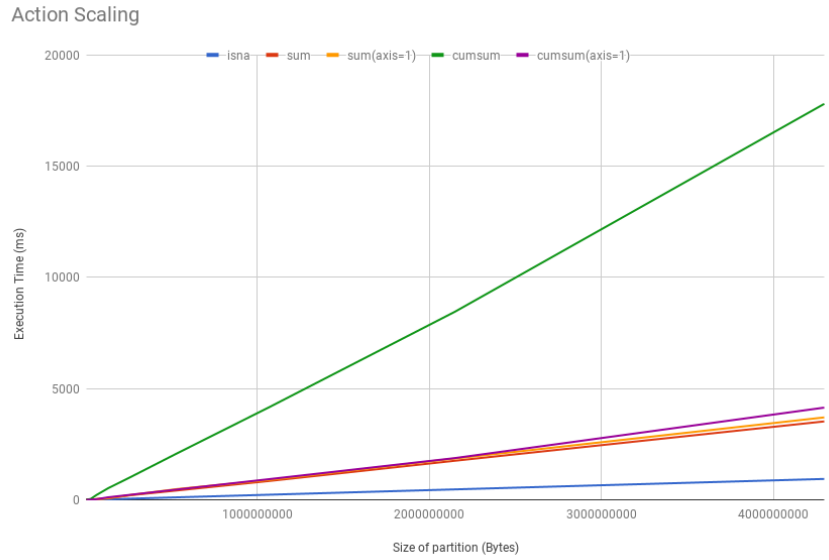

Figure 5.4: Action operation scaling versus number of bytes to concatenate. All operations examined here scaled linearly, which simplified the modeling logic.These operations were tested across 5 runs each.

5.5 Estimation Algorithm

The model uses data collected from the system in a file `params.dat`, which holds the performance metrics for the specific system and environment and which is loaded when the Ray DataFrames library is loaded. The model then proposes a range of possible partition sizes in both dimensions, and finds the partitioning split that results in the smallest estimated execution time, as shown in the abbreviated code in 5.1. For each proposed split, the optimizer makes sure to consider any possible performance penalties for coercing to that split from the current partition split, such as requiring a condense step, exploding partitions on the columns, or limiting utilization by under-parallelizing the system. This routine is called by any functions with partition optimization benchmarked and enabled, and those functions use a special remote function deployment mechanism that allows for the shuffling of the correct partitions to the correct tasks, as shown in the abbreviated code in 5.2.

```
def _optimize_partitions(in_dims, dsize, fname):
    in_rows, in_cols = in_dims

    def est_time(split):
        # Estimate the runtime of the proposed split considering
        # coercing from in_dims -> split, type of operation, etc.
        ...

    # For axis-dependent operations, one axis may be anchored at only
    # one partition, for correctness reasons
    candidate_rows = generate_candidate_rows(in_rows)
    candidate_cols = generate_candidate_cols(in_cols)
    candidate_splits = list(product(candidate_rows, candidate_cols))
    times = [(split, est_time(split)) for split in candidate_splits]
    split = min(times, key=lambda x: x[1])[0]
    return split
```

Listing 5.1: Performance Esimation and Optimization Logic

```
def example_func(self):
    block_rows, block_cols = dims = block_partitions.shape
    opt_rows, opt_cols = _optimize_partitions(dims, self.dsize, "example_func")
    result = np.empty((opt_rows, opt_cols), dtype=object)

    for i in range(opt_rows):
        for j in range(opt_cols):
            # Determine blocks to send to this task
            blocks = ...
            # Determine shape of sent blocks
            block_dims = ...
            result[i, j] = _deploy_func.remote(func, (i, j), block_dims, *blocks)

    return DataFrame(block_partitions=result)
```

Listing 5.2: Example function using Partition Optimization

5.6 Optimizer Deployment

In order to robustly collect and utilize system performance parameters, our prototype optimizer initially runs a set of benchmark diagnostics on their system. Since our userbase may utilize our tool for a diverse set of use cases on a a diverse range of hardware, it is important for us to fine tune our partitioning strategies for each system, automatically. The script does a number of smale-scale tests on loading, re-partitioning, and action runtimes to accurately fit the optimizer's internal prediction models for the various computational stages at runtime. Effectively, this allows our library to collect system performance parameters, such as CPU processing speed, memory bandwidth, disk read speeds, and IPC latency, indirectly, which is simpler to manage. Likewise, due to the inevitable error that arises from runtime prediction, our measurements allow for fewer assumptions compared to estimating runtime from system base parameters, allowing for more accurate predictions.

Chapter 6

Evaluation

In this section, we examine the performance of Ray DataFrames performance compared to Pandas under naive worst-case partitioning as well as optimized partitioning. Performance results are shown with timed microbenchmarks for the load and action stages, as well as a fully optimized workflow with selected functions. Experiments were done on Amazon Web Services on m4.2xlarge instances[1]. Specifications are listed with results.

6.1 Partition Optimization Benchmarks

All tests for partition operation benchmarking were performed by repeating the tested task over a set of dummy dataframes consisting of numeric data, in which the row dimension was much larger, usually 1000x or more, than the column dimension. This was done to emulate the size of a typical dataframe, where the number of records generally overshadows the number of datapoints per record.

Load

For the load stage, we benchmark `ray.dataframe.read_csv`, since it is the most common filetype used with Pandas analyses for most users. We observe in Fig. 6.1 how the performance of reading from disk scales well when increasing the number of partitions to load into, up to the number of cores of the system. The runtime then increases after, decreasing again until reaching another multiple of the number of cores. As expected, the runtime improvements overtime diminish over time with more partitions. Also worth noting, the additional overhead to convert the ingested row partitions into block partitions was negligible under a reasonable number of column partitions, and as such suggests that the best strategy would be to partition the dataframe into a reasonable set of block partitions based on the size of data used.

[1] 8 vCPUs, 32 GB of memory, https://aws.amazon.com/ec2/instance-types/

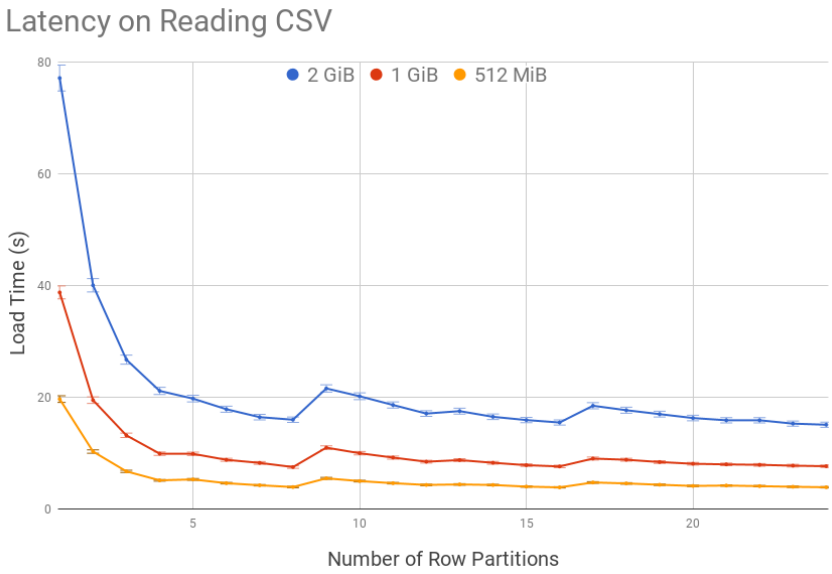

Figure 6.1: `read_csv` Scaling. Each of the upticks occur at a multiple of the number of cores. For reference, Pandas load times are 75s , 38s, 18s for 2 GiB, 1 GiB, and 512 MiB, respectively

Actions

As mentioned in Chapter 5 each DataFrame operation can be classified into a type of archetype, and a subset of the action archetypes are examined separately, with a specific function representing the archetype. All figures displayed in this section represent tests undertaken with a 2 GiB dataframe, with dimensions 1 million rows by 256 columns consisting of 64-bit integers.

Applymap

For the "applymap" archetype, we revisit the performance of `df.isna` from the previous section, with the figure reproduced here. Again, we observe on the left in Fig. 6.2 that the performance of the naive partitioning scheme follows a two-axis fan-out, with 1-by-1 partitioning performing badly and with performance improving. The performance follows a convex function with an inflection point around 8 total partitions, in which further partitions solely incur overhead due to superfluous remote tasks.

With partition optimization turned on, we compare observe the differences on the right of Fig. 6.2. We notice the much more robust performance of the function with respect to the inputs partitioning splits, especially with bad partitioning cases such as 1-by-1 initial partitions. Of concern is the performance for highly partitioned dataframes, such as those in the 16-by-16 case, in which performance stays near the same or suffers slightly. This performance anomaly can briefly be explained as the cost of additional tasks in Ray was

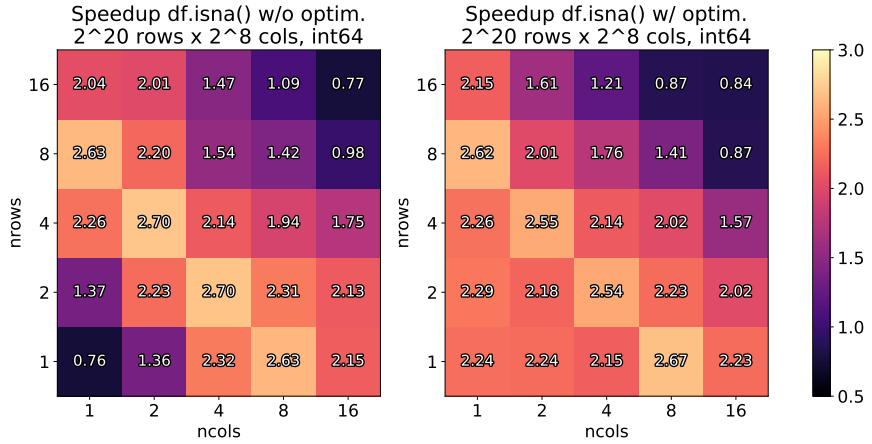

Figure 6.2: `df.isna` comparison. On the left, without partition optimization, compared with on the right, with partition optimization. The values represent the speedup over Pandas.

higher than expected, even though it was lower than the cost of condensing partitions to reduce superfluous task overhead.

Axis Reductive

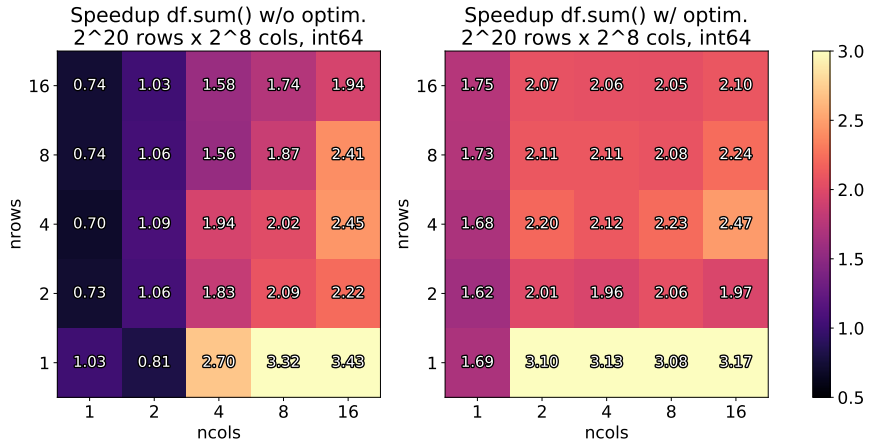

Figure 6.3: `df.sum` comparison. On the left, without partition optimization, compared with on the right, with partition optimization. The values represent the speedup over Pandas.

For the "axis-reduce" archetype, we examine the performance of `df.sum`, a function which returns a series with the summed values across an axis of the dataframe. We again compare the runtime performances in Fig. 6.3 for reducing across rows. For this function archetype, we notice a stark difference in performance with respect to the axis of operation, namely due to the default operation of Ray DataFrames, which will reconstruct an entire

axis of block partitions into row or column partitions, in order to guarantee concordance with Pandas operation. For operations such as `df.sum` or other axis reduction functions, operations are only performed after the subsequent blocks are concatenated together. Contrast this with other frameworks, which may perform a two-part reduce to combine the results from each individual block.

We again demonstrate how partition optimization generates much more robust results. We notice that the largest gains in performance are from poorly initially partitioned dataframes, which can be explained by the optimizer attempting to coerce the partitions to exploit better parallelism. As an example, for `df.sum(axis=0)`, which implies a sum across all rows, starting with 8 row and 1 column partition dimensions, the naive implementation will first reduce to 1 row partition, and then act on the single concatenated partition. In the optimized form, the optimizer will first explode the columns into 8 partitions, then reduce the row partition number to 1, in order to best exploit multi-core parallelism. Since individual column partitions are fully parallelizable under a row-reductive operation, and since exploding is a relatively cheap operation, we observe an almost 2.5x speedup in the example case specifically.

Cumulative

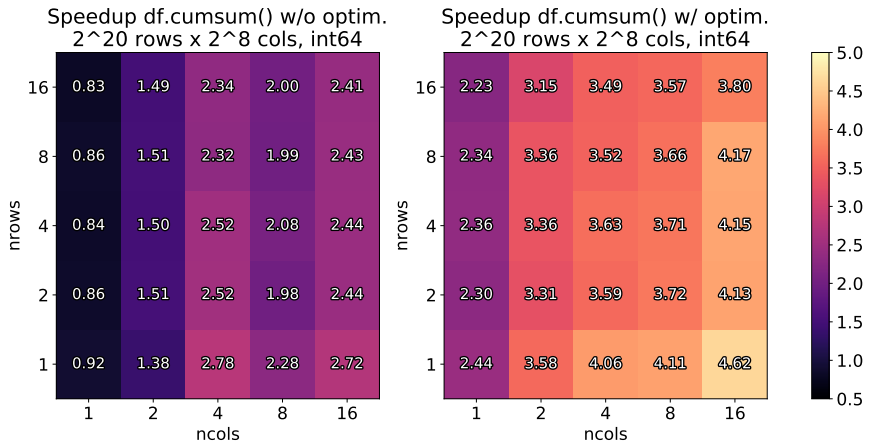

Figure 6.4: `df.cumsum` comparison. On the left, without partition optimization, compared with on the right, with partition optimization. The values represent the speedup over Pandas.

Lastly, for the "cumulative" archetype, we examine the performance of `df.cumsum`, a function which returns a new dataframe with entries corresponding to the cumulative sum of all values up to that entry in the previous dataframe, across an axis of the dataframe. Although this function is similar to the `df.sum` function shown above, we also examine `df.cumsum` to demonstrate the performance of Ray DataFrame with axis-variant functions that return a separate dataframe. We again compare the runtime performances in Fig. 6.4 for reducing across rows.

# Columns	Task Mean	Task SD	Critical Path	Total Mean	Total SD	Z-Score
16	1000	131.37	**2382**	2000	185.78	**2.06**
32	408	71.52	**1850.23**	1632	143.04	**1.51**
48	204	46.61	**1299.14**	1212	114.18	**0.66**

Table 6.1: Straggler Task Variance on `df.cumsum`. With finer-grained tasks critical path Z-Score decreases[2].

For `df.cumsum` we again notice that the largest gains in performance are from poorly initially partitioned dataframes, with better relative speedups compared to `df.sum`. We also see a large improvement in pre-well partitioned dataframes, with 1-by-16 partitioning experiencing almost a 2x speedup from the partition optimization. Examining this closer, this can primarily be explained by straggler variance. Manually testing various partitioning schemes, as we see in Table 6.1, the Z-score deviation from the critical path decreases with more partitioning, causing lower execution times despite partitioning that already exceeds the theoretical maximum parallelism ratio. This suggests the rationale for our variance cost estimator is correct, evident for computationally expensive tasks with larger variances.

6.2 Workflow Benchmarks

Apart from single-function microbenchmarks, we examined a small number of typical data science and exploratory data analysis workloads. While we initially planned on benchmarking performance versus Pandas on kernels and notebooks from Kaggle, this soon proved impossible as most of the datasets in the range suitable for Ray DataFrames, that is, larger than 50 MB, were locked behind competition terms and conditions. Most of the other datasets examined were too small, many in the sub-1 megabyte range, and these notebooks would perform faster on Pandas. Instead, we opted to use a separate dataset and generate a candidate EDA workflow to benchmark with Pandas.

For this workflow, we identified commonly used Pandas functions by scraping Kaggle kernels, with a candidate testing function per category of function from section 4.3. We used data from MLB StatCast [17] from past baseball seasons, as this data provided a good mix of string, numeric and categorical data, and was a suitable size for testing (each season's worth of events was roughly 250 MB on disk, 500 MB in memory).

From the results shown in Fig. 6.5, we can see some speedups from functions such as reading from disk and describing the dataset, while many of the other functions currently favor Pandas. Notable from this bunch are selecting a series from a dataframe as well as subsetting rows from a dataframe using a series. While these are observed as very frequently

[2]Values were collected in milliseconds from the Ray UI. Total path statistics represent the estimated **total runtime** statistics based on measured **individual task** statistics the number of tasks per executor as the number of samples, assuming normally distributed samples. We use similar assumptions when estimating maximum variance deviation for the cost model

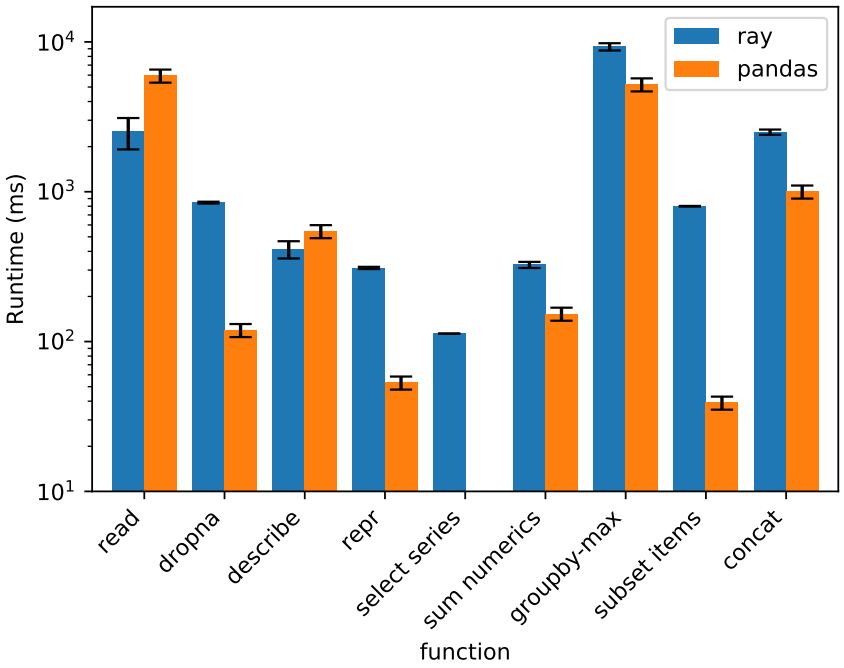

Figure 6.5: Common workflow function comparison. All operations were tested 5 times each. Note that for "select series", the Pandas operation consistently ran in under 200 μs.

used on Kaggle kernels, due to the need to extract data series's for use with plotting frameworks, our current implementation struggles with these operations since we fall back onto Pandas' series implementation. This requires a full computation step whenever extracting a series, while Pandas series extraction produces a view on the original data. A distributed series implementation is something we aim to add in the future, to help alleviate performance with series from dataframes (See section 8.1).

Chapter 7

Related Works

7.1 Numerical Analysis Tools

Ray DataFrames is designed to be an evolution to existing data analysis toolkits, the most influential being Pandas [18], for acting as a basis to emulate and also for partially being used as the underlying computation framework for Ray DataFrames. Many other frameworks exist with similar functionality, such as the R [23] and S [7] languages as well as the dplyr [35] library for R. As all these frameworks are built in their primary forms for single-core, single-node systems, they do not scale well on modern, multi-core systems. The underlying execution architecture was also influenced by NumPy and other libraries in the SciPy language, as careful steps were undertaken to ensure performant execution with the libraries used.

7.2 Parallelized Data Analysis Tools

A large number of commercial and academic projects have examined improving performance of distributed dataframe, matrix, or array operations in Python, such as PySpark [4] and Dask [26]. Outside of the Python ecosystem, systems such as Spark's R [34] and SQL [4] implementations, Apache Hive [32] and Pig [20], and DryadLINQ's [37] SQL integration have attempted to adapt MapReduce architectures into user-friendly interfaces. However, these interfaces all optimize towards bulk pipeline execution with lazy semantics, which work well for workloads aimed at solely at processing large quantities of data, such as Extract-Transform-Load (ETL), but fall short with respect to interactive and fine grained workloads. Namely, the MapReduce [10] architecture for which most of these projects are based on performs very well for workloads that fit into the MapReduce bulk synchronous parallel paradigm, but is incompatible with tasks that aim to make fine-grained modifications of existing data. Likewise, the lazy execution paradigm, which is a defining feature of frameworks such as Spark and Tensorflow [1], is not conducive to interactive workloads due to the frequency and process of EDA.

The Rmpi and SNOW packages for R share very many features and design choices with Ray DataFrames, allowing for direct parallelization of R workflows using MPI as an underlying execution framework. However, both frameworks force the end user to be much more involved in parallelizing existing dataframes and code, requiring at least a cursory understanding of the MPI framework. Ray DataFrames is aimed at making parallel execution as opaque and painless to the user as possible. Likewise, Ray DataFrames' usage of Ray as its underlying execution framework adds in support for fault tolerance and a high level abstraction for shared memory with the Plasma Store.

7.3 Cost-based Optimization

Our work in data partitioning estimation is closely related to existing work on static analysis and estimation for data partitioning [5, 12]. Our estimation models, model training, and optimization techniques very closely mirror the existing work. We also adapt many of the techniques for past work on adapting benchmark results for performance estimation [27] and regression-based techniques for benchmark estimation [6]. While most of the existing techniques involve research on compiler partitioning optimization [11], we extend the existing techniques for estimation and optimization at runtime, for dynamic partitioning. Likewise, while past research on partition optimization mostly involved optimizing over communication-based parallel frameworks on non-uniform or distributed contexts, our system uses different assumptions, namely much reduced communication overhead, with a higher focus on dynamics related to shared memory usage across multiple workers. We use those assumptions and dynamics to better inform our estimation modeling in this domain.

More recent research has examined cost-based optimization techniques for MapReduce programs [13], transient view query plans [30], and general database query execution [8]. Although this work is related in domain and scope to Ray DataFrames, the optimization techniques proposed are mostly examining optimizing query execution plans and orderings, while we mostly examine localized partition optimization. Importantly we use a similar technique in acting on estimated performance parameters, and there is certainly room for future work to intertwine cost-based optimization for partitioning and query planning.

Chapter 8

Future Work

8.1 API Expansion

Ray DataFrames is still an early stage project, and the functions implemented in this report only cover the dataframe API exported by Pandas at a surface level. We plan on expanding the API coverage for Ray DataFrames to other data structures in Pandas, especially Series and Multi-level Indexes, which are features frequently used for plotting or grouping analyses. Likewise, some components of the dataframe API are only semi-covered, or are done so in a manner that emphasizes correctness over efficiency, such as performing a collect-reduce instead of a two-part or tree-based reduce for axis-based arithmetic functions.

8.2 Expanding past Single-Node Operation

Currently, although the Ray DataFrames library is built atop Ray, it is constrained to a single node. While it would be relatively easy to simply enable cluster operation by allowing the user to connect to an existing Ray cluster, many assumptions are currently made on top of single-node operation. The biggest assumption broken with cluster operation is the locality and extremely low latency of Plasma Store accesses, which currently do not take into account network latency required to move objects between machines. Likewise, remote task dispatch requires much more complex articulation with regards to the placement and distribution of partitions across nodes, as currently the architecture assumes homogeneous and uniform workers.

Another area to support single- and multi-node operation would be the capability to spill to disk. This functionality is currently support by frameworks such as Spark [4] and Dask [26], which enables those frameworks to scale to larger-than-memory datasets. Currently, the Plasma Store does not support manual eviction, let alone spill to disk capability, so work will have to be done to determine the best place in the stack to implement a mapping from Object IDs to locations, whether on disk or in memory.

8.3 Look-ahead query optimization

As mentioned in Chapter 5, the partition optimization model only considers the current operating task, without regards to future operations. While look-ahead optimization for lazy-evaluation systems is much simpler since all query planning can be done at the collection stage, the same cannot be said for eager-evaluation systems since the model cannot be certain on the user's next operation. One possible strategy for look-ahead optimization would include reinforcement learning-based prediction [31]. In particular, model-based RL may be a good fit to best estimate both performance dynamics of new functions and MDP-based state transitions would be a good fit for estimating the best partitioning splits to optimize for global execution.

Partition optimization is also only a small part of the possible scope of query optimization for interactive workflows. We colloquially call this process "eager query planning" or "eager query optimization", which would adapt database query planning techniques [8] as well as execution akin to microprocessor branch prediction [28] to best optimize interactive and semi-interactive workloads.

Chapter 9

Conclusion

In this report we presented Ray DataFrames, an evolutionary iteration of existing exploratory data analysis tools. We introduced the mechanisms and techniques performed to parallelize Pandas and existing Pandas workflows, allowing for significant speedups over Pandas in multi-core, modern systems. We also demonstrated techniques for cost-based partition optimization with performance prediction to provide fast and robust operation. Although some popular operations are currently slower than regular Pandas, namely operations that require constructing views of data, we demonstrate a promising foundation for parallel execution, with optimized functions achieving up to 4.6 times speedup on modestly-sized datasets.

The increasing popularity of data science and inflation of datasets produce strain on existing data science and data analysis tools. To address these issues, Ray DataFrames presents a toolkit for scalable exploratory data analyses, providing parallel execution on large datasets previously deemed too large to manage, with a familiar interface and operational semantics. Ray DataFrames places emphasis on optimizing eager evaluation with interactive environments favorable for data scientists or novice users who may not have the same experience as systems engineers.

Bibliography

[1] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[2] Ganesh Ananthanarayanan et al. "Effective Straggler Mitigation: Attack of the Clones." In: *NSDI*. Vol. 13. 2013, pp. 185–198.

[3] *Apache Arrow*. `https://arrow.apache.org/`.

[4] 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.

[5] Vasanth Balasundaram et al. "A static performance estimator to guide data partitioning decisions". In: *ACM Sigplan Notices*. Vol. 26. 7. ACM. 1991, pp. 213–223.

[6] Bradley J Barnes et al. "A regression-based approach to scalability prediction". In: *Proceedings of the 22nd annual international conference on Supercomputing*. ACM. 2008, pp. 368–377.

[7] Richard A Becker. "A brief history of S". In: *cahier de recherche, AT&T Bell La* (1994).

[8] Surajit Chaudhuri. "An overview of query optimization in relational systems". In: *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM. 1998, pp. 34–43.

[9] *Dask DataFrame Performance Tips*. Mar. 2018. URL: `https://dask.pydata.org/en/latest/dataframe-performance.html`.

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

[11] Manish Gupta and Prithviraj Banerjee. "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers". In: *IEEE Transactions on Parallel and Distributed Systems* 3.2 (1992), pp. 179–193.

[12] Philip J Hatcher et al. "Data-parallel programming on MIMD computers". In: *IEEE Transactions on Parallel and Distributed Systems* 2.3 (1991), pp. 377–383.

[13] Herodotos Herodotou and Shivnath Babu. "Profiling, what-if analysis, and cost-based optimization of mapreduce programs". In: *Proceedings of the VLDB Endowment* 4.11 (2011), pp. 1111–1122.

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

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

[16] Dan Kopf. *Meet the Man behind the Most Important Tool in Data Science*. qz.com/1126615/the-story-of-the-most-important-tool-in-data-science/. Article. 2017.

[17] Marcos Lage et al. "Statcast dashboard: Exploration of spatiotemporal baseball data". In: *IEEE computer graphics and applications* 36.5 (2016), pp. 28–37.

[18] 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.

[19] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *CoRR* abs/1712.05889 (2017). arXiv: 1712.05889. URL: http://arxiv.org/abs/1712.05889.

[20] 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.

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

[22] *Plasma In-Memory Object Store*. https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store/.

[23] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2013. URL: http://www.R-project.org/.

[24] *Ray Project*. https://github.com/ray-project/ray. 2018.

[25] David Robinson. *Why is Python growing so quickly?* https://stackoverflow.blog/2017/09/14/python-growing-quickly/. Blog. 2017.

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

[27] Rafael H Saavedra and Alan J Smith. "Analysis of benchmark characteristics and benchmark performance prediction". In: *ACM Transactions on Computer Systems (TOCS)* 14.4 (1996), pp. 344–384.

[28] James E Smith. "A study of branch prediction strategies". In: *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press. 1981, pp. 135–148.

[29] *Spark SQL, DataFrames and Datasets Guide*. URL: https://spark.apache.org/docs/2.2.0/sql-programming-guide.html#bucketing-sorting-and-partitioning.

[30] Subbu N Subramanian and Shivakumar Venkataraman. "Cost-based optimization of decision support queries using transient-views". In: *ACM SIGMOD Record*. Vol. 27. 2. ACM. 1998, pp. 319–330.

[31] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[32] 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.

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

[34] Shivaram Venkataraman et al. "Sparkr: Scaling r programs with spark". In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 1099–1104.

[35] Hadley Wickham and Romain Francois. "dplyr: A grammar of data manipulation". In: *R package version 0.4* 3 (2015).

[36] Wikipedia contributors. *Block matrix — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-May-2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Block_matrix&oldid=821643233.

[37] Yuan Yu et al. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language." In: *OSDI*. Vol. 8. 2008, pp. 1–14.