O'REILLY®

Dask

The Definitive Guide

Scalable Python Data Science with Dask

Early Release

RAW & UNEDITED

Matthew Rocklin,
Matthew Powers
& Richard Pelgrim

Dask: The Definitive Guide

Scalable Python Data Science with Dask

Matthew Rocklin, Matthew Powers, and Richard Pelgrim

Dask: The Definitive Guide

by Matthew Rocklin, Matthew Powers, and Richard Pelgrim

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Virginia Wilson and Jessica Haberman

Production Editor: Ashley Stussy

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

September 2023: First Edition

Revision History for the Early Release

- 2022-05-25: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098117146 for release details.

978-1-098-11708-5

[FILL IN]

Chapter 1. Understanding the Architecture of Dask DataFrames

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 3rd chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

Dask DataFrames allow you to scale your pandas workflows. Dask DataFrames overcome two key limitations of pandas:

- pandas cannot run on datasets larger than memory
- pandas only uses one core when running analyses, which can be slow

Dask DataFrames are designed to overcome these pandas limitations. They can be run on datasets that are larger than memory and use all cores by default for fast execution. Here are the key Dask DataFrame architecture components that allow for Dask to overcome the limitation of pandas:

- Partitioning data
- Lazy execution
- Not loading all data into memory at once

Let's take a look at the pandas architecture first, so we can better understand how it's related to Dask DataFrames.

You'll need to build some new mental models about distributed processing to fully leverage the power of Dask DataFrames. Luckily for pandas programmers, Dask was intentionally designed to have similar syntax. pandas programmers just need to learn the key differences when working with distributed computing systems to make the Dask transition easily.

pandas Architecture

pandas DataFrames are in widespread use today partly because they are easy to use, powerful, and efficient. We don't dig into them deeply in this book, but will quickly review some of their key characteristics. First, they contain rows and values with an index.

Let's create a pandas DataFrame with `name` and `balance` columns to illustrate:

```
import pandas as pd
df = pd.DataFrame({"name": ["li", "sue", "john", "carlos"], "balance":
[10, 20, 30, 40]})
```

This DataFrame has 4 rows of data, as illustrated in Figure 1-1.

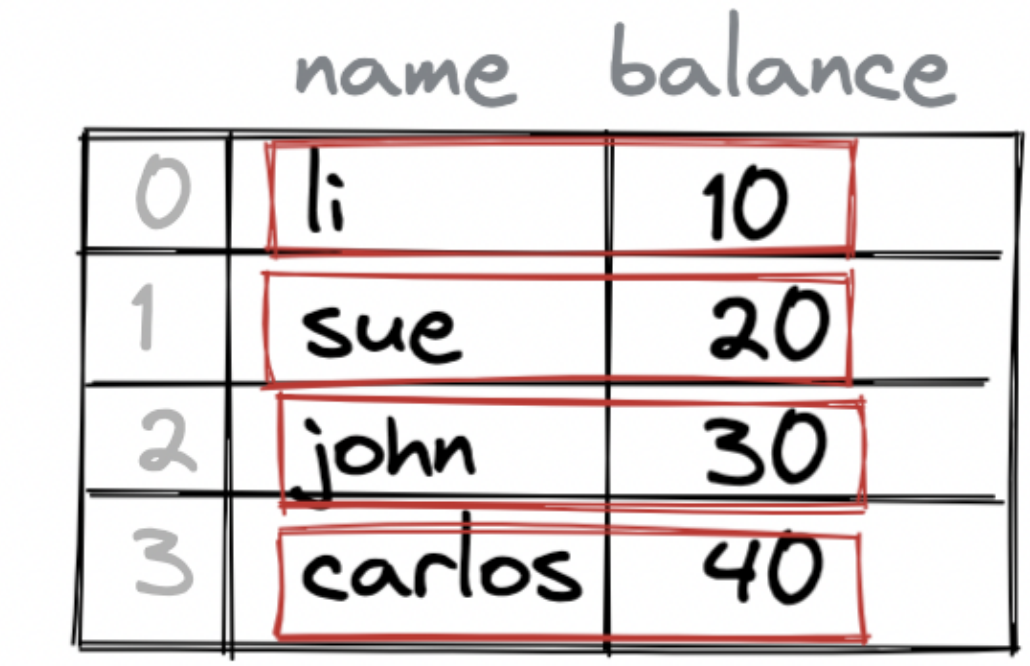

Figure 1-1. pandas DataFrame with four rows of data

The DataFrame in Figure 1-2 also has an index.

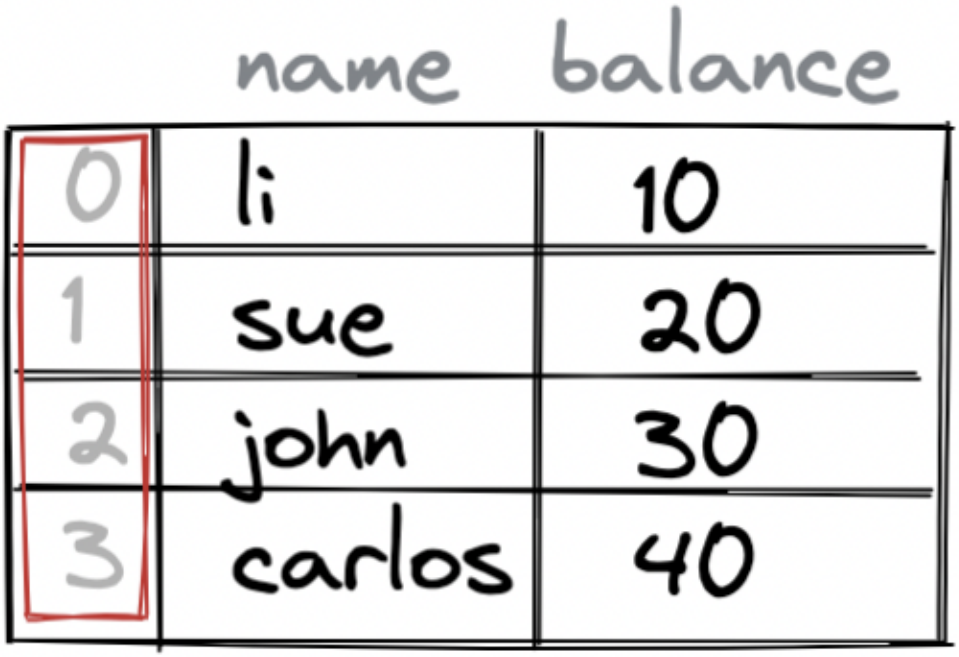

Figure 1-2. pandas DataFrame has an index

pandas makes it easy to run analytical queries on the data. It can also be leveraged to build complex models and is a great option for small datasets, but does not work well for larger datasets. Let's look at why pandas doesn't work well for bigger datasets.

pandas Limitations

pandas has two key limitations:

- Its DataFrames are limited by the amount of computer memory
- Its computations only use a single core, which is slow for large datasets

pandas DataFrames are loaded into the memory of a single computer. The amount of data that can be stored in the RAM of a single computer is limited to the size of the computer's RAM. A computer with 8 GB of memory can only hold 8 GB of data in memory. In practice, pandas requires the memory to be much larger than the dataset size. A 2 GB dataset may require 8 GB of memory for example (the exact memory requirement depends on the operations performed and pandas version). Figure 1-3 illustrates the types of datasets pandas can handle on a computer with 16 GB of RAM.

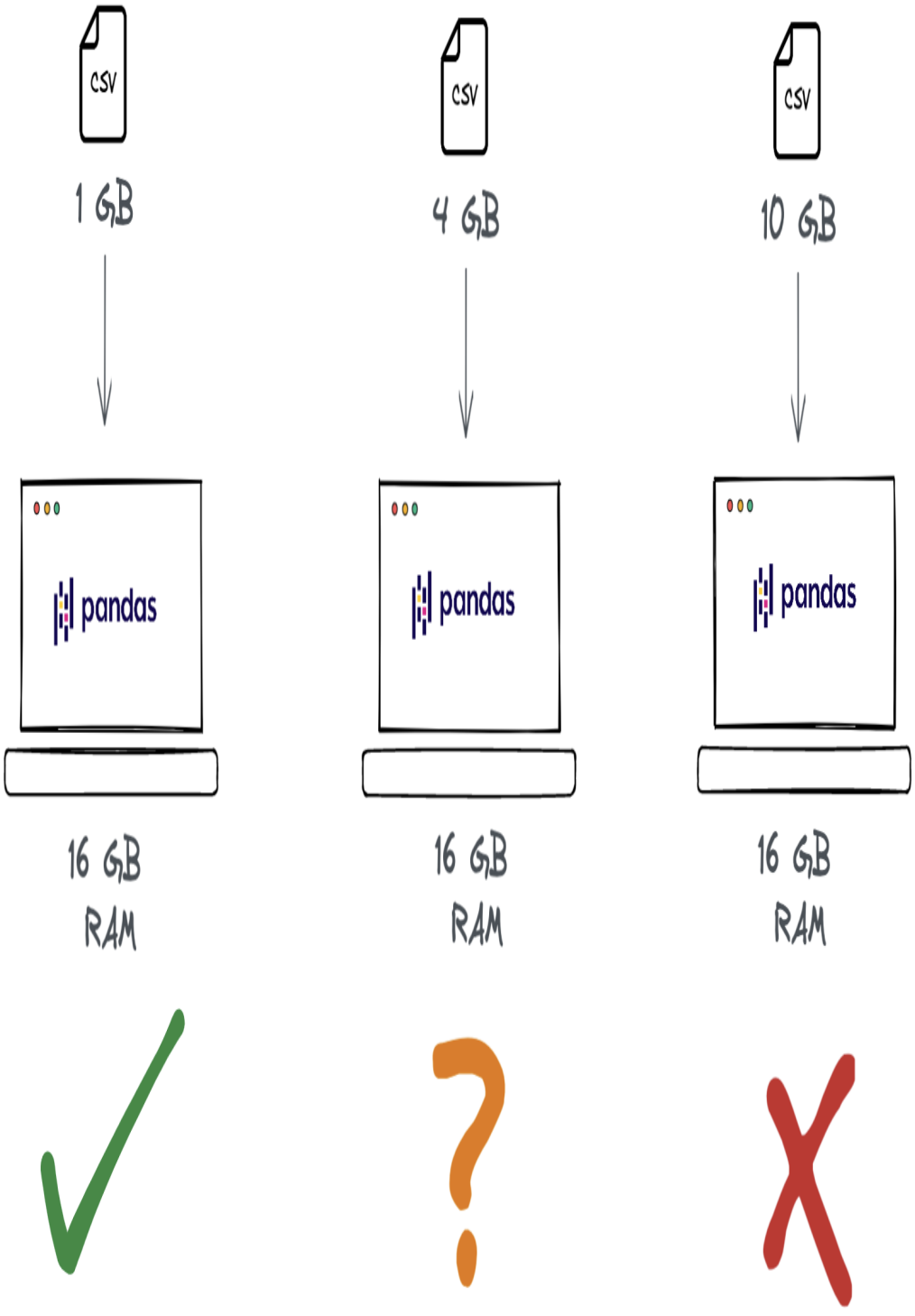

Figure 1-3. Dataset sizes pandas can handle on a computer with 16 GB of RAM

Furthermore, pandas does not support parallelism. This means that even if you have multiple cores in your CPU, with pandas you are always limited to using only one of the CPU cores at a time. And that means you are regularly leaving much of your hardware potential untapped (see FIgure 3-4).

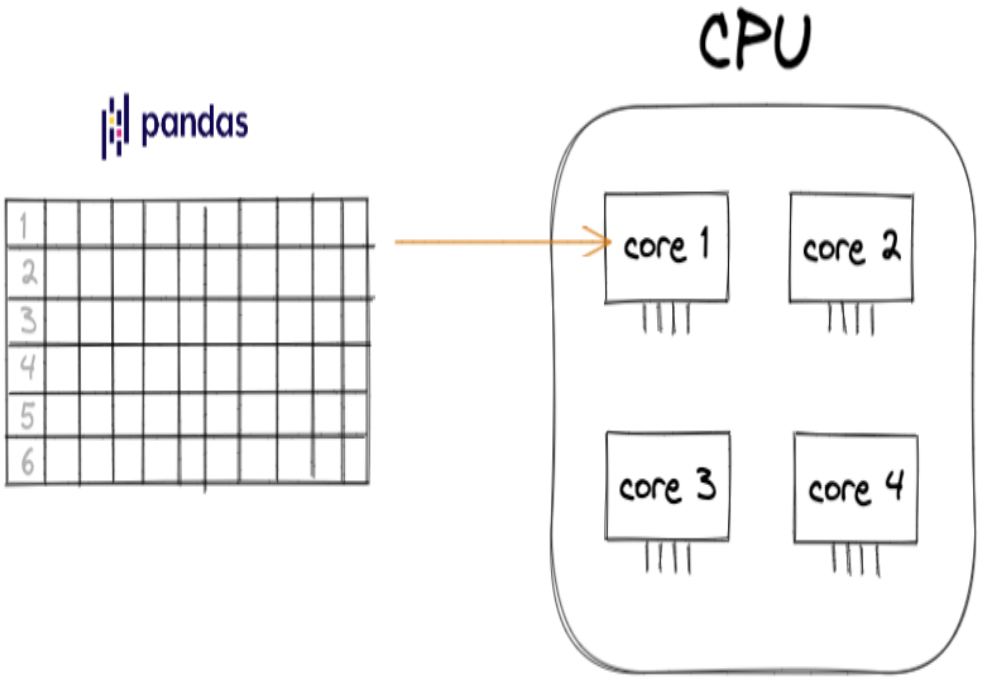

Figure 1-4. pandas only uses a single core and don't leverage all available computation power

Let's turn our attention to Dask and see how it's architected to overcome the scaling and performance limitations of pandas.

How Dask DataFrames Differ from pandas

Dask DataFrames have the same logical structure as pandas DataFrames, and share a lot of the same internals, but have a couple of important architectural differences. As you can see in Figure 1-5, pandas stores all data in a single

DataFrame, whereas Dask splits up the dataset into a bunch of little pandas DataFrames.

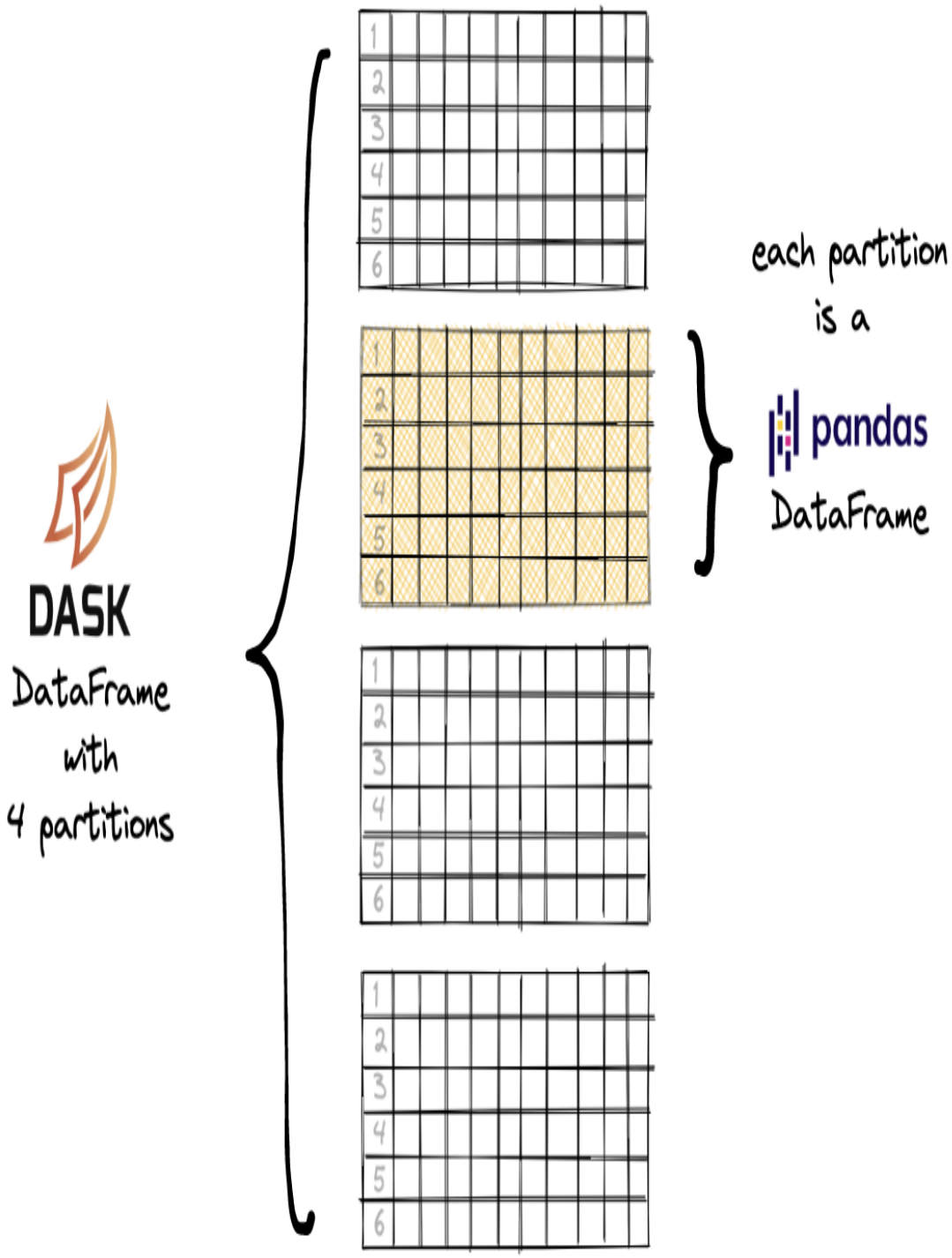

Figure 1-5. Each partition in a Dask DataFrame is a pandas DataFrame

Suppose you have a pandas DataFrame with the following contents:

```
        col1    col2
0       a       1
1       b       2
2       c       3
3       d       4
```

This pandas DataFrame can be converted to a Dask DataFrame, which will split up the data into a bunch of smaller partitions. Each partition in a Dask DataFrame is a pandas DataFrame. A Dask DataFrame consists of a bunch of smaller pandas DataFrames.

Similar to a pandas DataFrame, a Dask DataFrame also has columns, values, and an index. Notice that Dask splits up the pandas DataFrame by rows.

Dask DataFrames coordinate many pandas DataFrames in parallel. They arrange many pandas dataframes split along the index.

Dask DataFrames don’t have to be in memory at once because the values are now split into many different pieces. Figure 1-6 shows how Dask DataFrames can load the pieces one at a time, allowing us to compute on datasets that are larger than memory.

Figure 1-6. Dataset sizes that Dask DataFrames can handle

Dask DataFrames can also be processed in parallel because the data is split into pieces, which often leads to faster processing. Figure 1-7 shows how each Dask DataFrame partition (which is just a pandas DataFrame) can be processed on a separate CPU core.

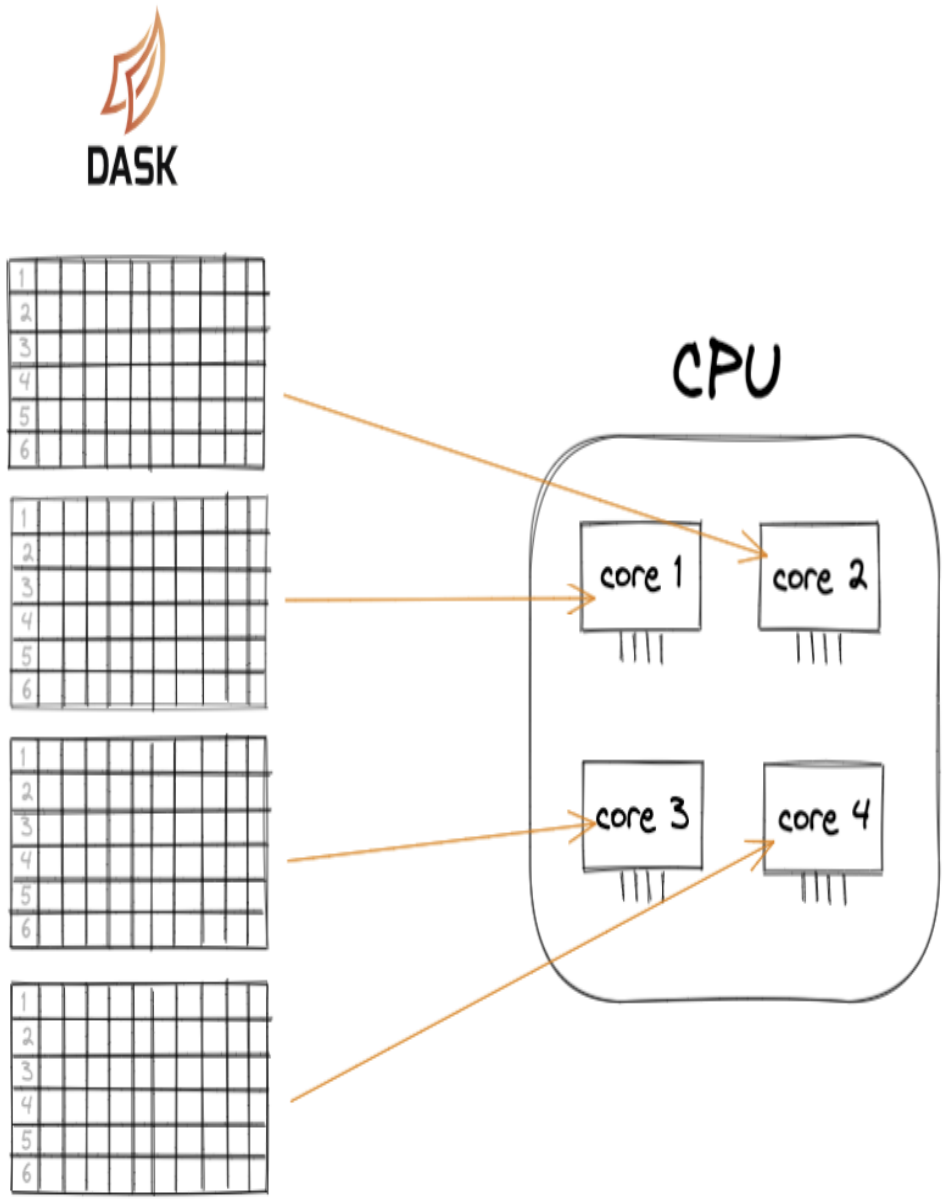

Figure 1-7. Dask DataFrames run computations with all available cores

However, because not all the data is in memory at once, some operations are slower or more complicated. For example, operations like sorting a DataFrame or finding a median value can be more difficult. See Chapter 4 for more information and best practices.

Example illustrating Dask DataFrame Architectural Components

Let's illustrate the architectural concepts discussed in the previous section with a simple code example. We'll create a pandas DataFrame and then convert it to a Dask DataFrame to highlight the differences.

Here's the code to create a pandas DataFrame with `col1` and `col2` columns:

```
import pandas as pd
df = pd.DataFrame({"col1": ["a", "b", "c", "d"], "col2": [1, 2, 3,
4]})
        col1    col2
0       a       1
1       b       2
2       c       3
3       d       4
```

Now convert the pandas DataFrame into a Dask DataFrame (`ddf`) with two partitions.

```
import dask.dataframe as dd
ddf = dd.from_pandas(df, npartitions=2)
```

The data in the Dask DataFrame is split into two partitions because we set `npartitions=2` when creating the Dask DataFrame.

Dask intentionally splits up data into different partitions, so it can run computations on the partitions in parallel. Dask's speed and scalability hinge on its ability to break up computations into smaller chunks and run them using all the computational cores available on a machine.

Figure 1-8 illustrates how the Dask DataFrame is split into two partitions, each of which is a pandas DataFrame:

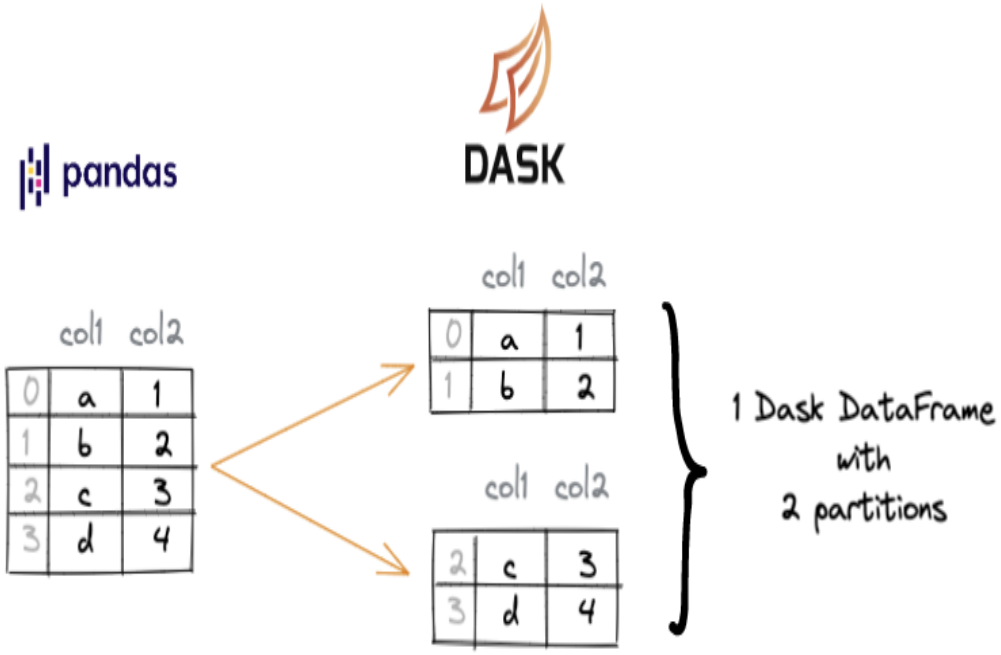

Figure 1-8. pandas DataFrame is split into Dask DataFrame partitions

Dask's architecuture of splitting up the data also allows for computations to be lazily executed.

Lazy Execution

Dask DataFrames use *lazy* execution, whereas pandas uses *eager* execution. Dask will put off running computations till the last minute in contrast with pandas, which executes computations immediately. This allows Dask to do two things pandas can't do:

1. Process datasets that are larger than memory
2. Collect as much data as possible about the computation you want to run and then optimize the computation for maximum performance.

Let's create a pandas DataFrame and run a filtering operation to demonstrate that it runs computations immediately (eager execution). Then let's run the same filtering operation on a Dask DataFrame to observe the lazy execution:

```
import pandas as pd
df = pd.DataFrame({"letter": ["a", "b", "c", "d"], "number": [10, 20,
30, 40]})
```

Filter the pandas DataFrame to only include the rows with a `number` value greater than 25.

```
df[df.number > 25]
        letter  number
2       c         30
3       d         40
```

pandas immediately executes the computation and returns the result.

Let's convert the pandas DataFrame to a Dask DataFrame with two partitions:

```
import dask.dataframe as dd
ddf = dd.from_pandas(df, npartitions=2)
```

Now let's run the same filtering operation on the Dask DataFrame and see that no actual results are returned:

```
ddf[ddf.number > 25]
```

Here's what's output:

```
Dask DataFrame Structure:
               letter   number
npartitions=2
             0  object   int64
             2     ...      ...
             3     ...      ...
Dask Name: loc-series, 8 tasks
```

Dask doesn't run the filtering operation computations unless you explicitly ask for results. In this case, you've just asked Dask to filter and haven't asked for results to be returned, and that's why the resulting Dask DataFrame doesn't contain data yet. pandas users find Dask's lazy execution strange at first, and it

takes them a while to get used to explicitly requesting results (rather than eagerly receiving results).

In this case, you can get results by calling the `compute()` method which tells Dask to execute the filtering operation and collect the results in a pandas DataFrame:

```
ddf[ddf.number > 25].compute()
        letter  number
2       c         30
3       d         40
```

Let's turn our attention to another key architectural difference between Dask DataFrames and pandas.

Dask DataFrame Divisions

The pandas index is key to many performant operations like time series operations, efficient joins, finding specific values quickly, and so on. Dask DataFrames don't store the entire pandas index in memory, but they do track index ranges for each partition, called *divisions*.

Figure 1-9 shows an example of a Dask DataFrame that's partitioned by month and divisions that are stored in the Dask DataFrame. Dask divisions track the starting index value of each partition as well as the ending index value of the last partition.

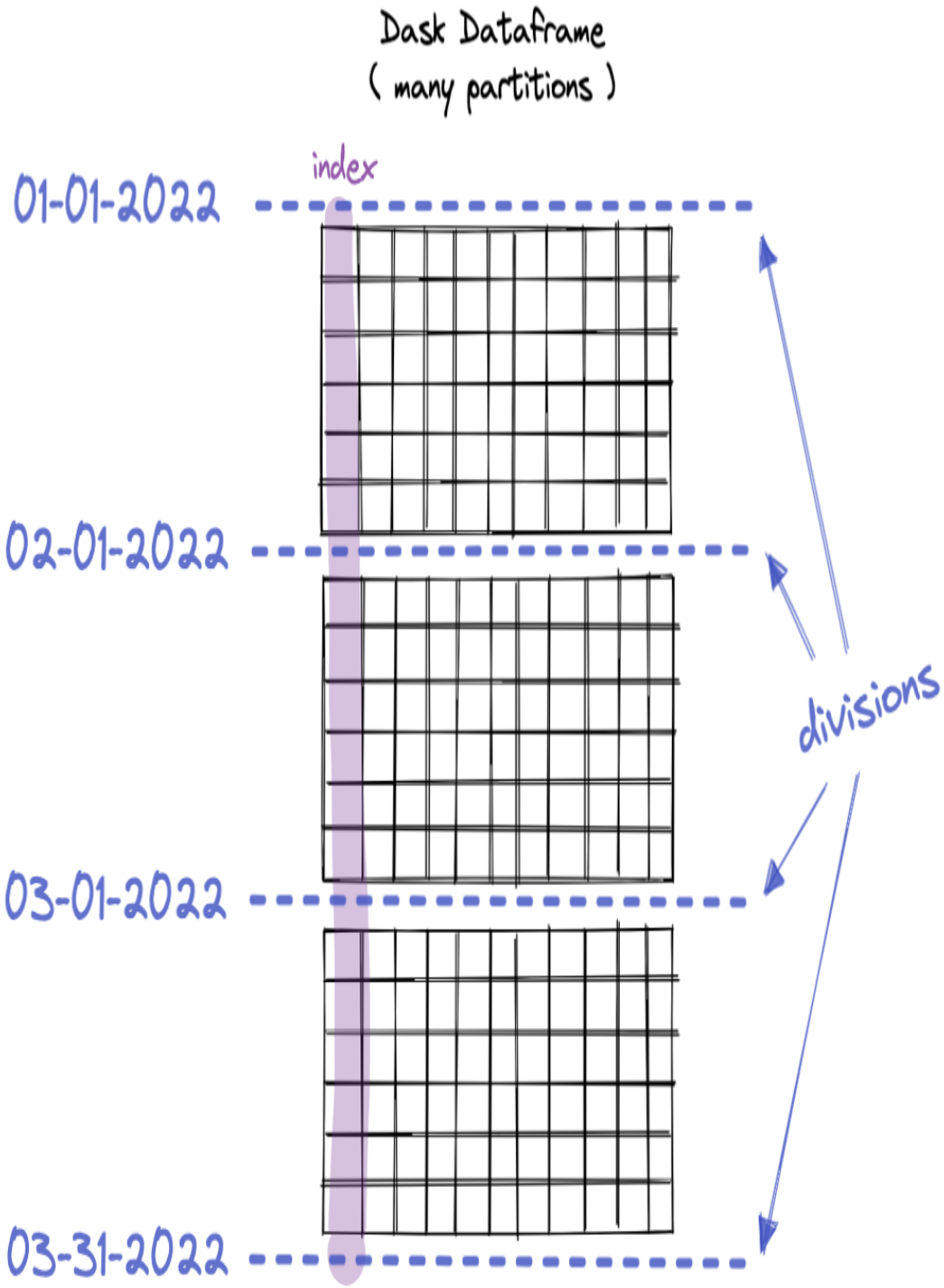

Figure 1-9. Dask DataFrame that's partitioned by month

Divisions are key to Dask DataFrames in much the same way that the pandas index is critical to pandas DataFrames. Chapter 4 will show you how good tracking of index/division information can lead to greatly improved performance.

Figure 1-10 looks at the same Dask DataFrame from earlier and explore the DataFrame divisions in more detail.

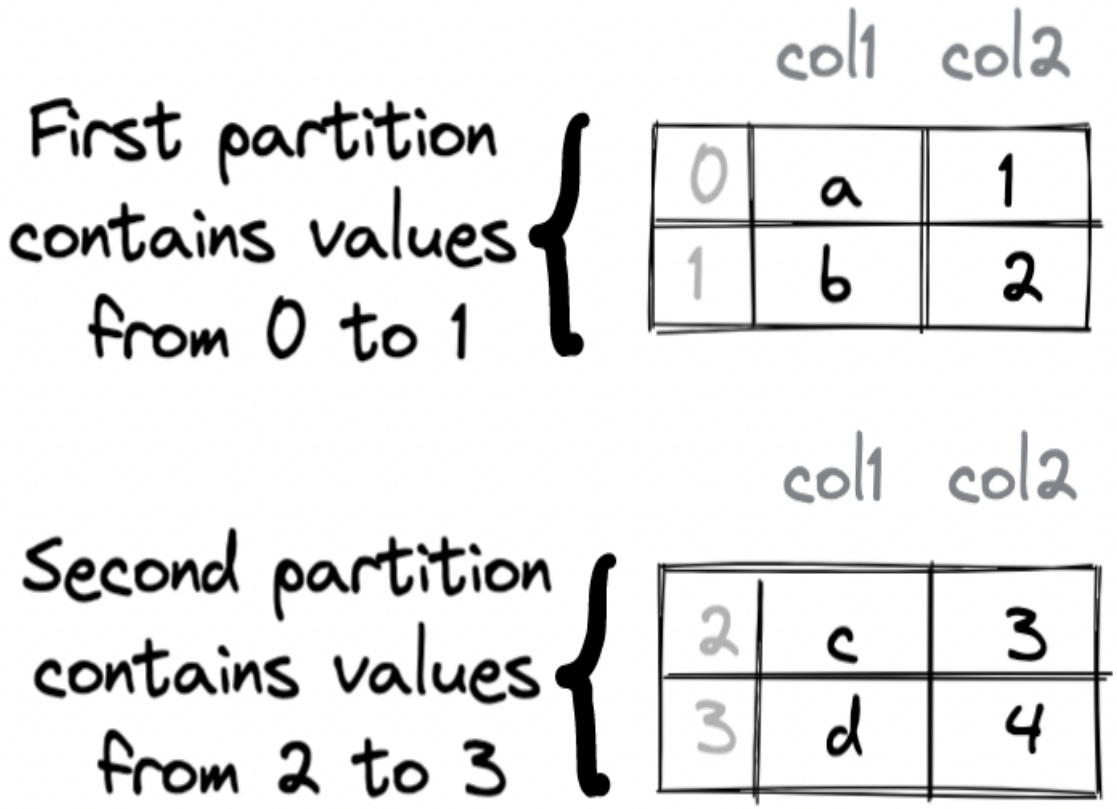

Figure 1-10. Dask DataFrame with divisions by partition

Notice that the first partition contains rows with index 0 and index 1, and the second partition contains rows with index 2 and index 3.

You can access the `known_divisions` property to figure out if Dask is aware of the partition bounds for a given DataFrame. `ddf.known_divisions` will return `True` in this example because Dask knows the partition bounds.

The `divisions` property will tell you the exact division bounds for your DataFrame:

```
ddf.divisions # (0, 2, 3)
```

Here's how to interpret the `(0, 2, 3)` tuple that's returned:

- The first partition contains index values that span from zero to two (not inclusive upper boundary)
- The second partition contains index values that span from two to three (inclusive upper boundary)

Suppose you ask Dask to fetch you the row with index 3. Dask doesn't need to scan over all the partitions to find the value. It knows that the row with index 3 is in the second partition from the `divisions` metadata, so it can narrow the search for index 3 to a single partition. This is a significant performance optimization, especially when there are thousands of partitions.

NOTE

[TIP] Knowing how to properly set the index and manage divisions is necessary when optimizing Dask DataFrame performance.

DASK DATAFRAMES DON'T LOAD ALL DATA INTO MEMORY BY DEFAULT

When you read data from a file into a pandas DataFrame, pandas will save all of the data in memory. This is fine for small datasets, but is a liability for larger datasets. Suppose you have a computer with 8 GB of RAM and would like to join two DataFrames, one which is 1 GB and another that's 0.25 GB. Those datasets easily fit in memory, so pandas will be able to easily store both DataFrames in memory and execute the computations.

pandas requirement to store all data in RAM becomes a problem when data sizes grow. Suppose you'd like to join one dataset that's 5 GB with another that's 4 GB using pandas. pandas isn't able to store 9 GB of data on a computer that only has 8 GB of RAM, so it'll error out and won't be able to perform this operation.

Dask does not store all the data in memory by default, so you can use Dask to process datasets that are bigger than the available memory. Let's look at a real example that demonstrates how Dask can run a query on a dataset that's bigger than the available RAM while pandas cannot.

pandas vs. Dask DataFrame on Larger than RAM Datasets

This section creates a larger than RAM dataset and demonstrates how pandas cannot run queries on this data, but Dask can query the data.

Let's use Dask to create 1,095 Snappy compressed Parquet files of data (58.2 GB) on your local machine in the *~/data/timeseries/20-years/parquet/* directory:

```
import os
import dask
home = os.path.expanduser("~")
ddf_20y = dask.datasets.timeseries(
    start="2000-01-01",
    end="2020-12-31",
    freq="1s",
```

```
    partition_freq="7d",
    seed=42,
)
ddf_20y.to_parquet(
    f"{home}/data/timeseries/20-years/parquet/",
    compression="snappy",
    engine="pyarrow",
)
```

Here are the files that are output to disk:

```
data/timeseries/20-years/parquet/
  part.0.parquet
  part.1.parquet
  …
  part.1095.parquet
```

Here's what the data looks like:

timestamp	id	name	x	y
2000-01-01 00:00:00	1008	Dan	-0.259374	-0.118314
2000-01-01 00:00:01	987	Patricia	0.069601	0.755351
2000-01-01 00:00:02	980	Zelda	-0.281843	-0.510507
2000-01-01 00:00:03	1020	Ursula	-0.569904	0.523132
2000-01-01 00:00:04	967	Michael	-0.251460	0.810930

There is one row of data per second for 20 years, so the entire dataset contains 662 million rows.

Let's calculate the mean value of the `id` column for one of the data files with pandas:

```
path = f"{home}/data/timeseries/20-years/parquet/part.0.parquet"
```

```
df = pd.read_parquet(path)
df["id"].mean() # 999.96
```

pandas works great when analyzing a single data file. Now let's try to run the same computation on all the data files with pandas:

```
import glob
path = f"{home}/data/timeseries/20-years/parquet"
all_files = glob.glob(path + "/*.parquet")
df = pd.concat((pd.read_parquet(f) for f in all_files))
```

Whoops! This errors out with an out of memory exception. Most personal computers don't have even nearly enough memory to hold a 58.2 GB dataset. This pandas computation will fill up all the computer's RAM and then error out.

Let's run this same computation with Dask:

```
ddf = dd.read_parquet(f"{home}/data/timeseries/20-years/parquet",
engine="pyarrow")
ddf["id"].mean().compute()# returns 1000.00
```

This computation takes 8 seconds to execute on a computer with 8 GB of RAM and four cores. A computer with more cores would be able to execute the Dask computation with more parallelism and run even faster.

As this example demonstrates, Dask makes it easy to scale up a localhost workflow to run on all the cores of a machine. Dask doesn't load all of the data into memory at once and can run queries in a streaming manner. This allows Dask to perform analytical queries on datasets that are bigger than memory.

This example shows how Dask can scale a localhost computation, but that's not the only type of scaling that Dask allows for.

Scaling Up vs Scaling Out

Dask DataFrame can scale pandas computations in two different ways:

- *Scale up:* Use all the cores of a computer rather than one core like pandas

- *Scale out:* Execute queries on multiple computers (called a cluster) rather than on a single computer

As previously discussed, pandas workflows only use a single core of a single machine. So pandas will only use one core, even if 8 or 96 are available. Dask scales up single machine workflows by running computations with all the cores, in parallel. So if Dask is run on a machine with 96 cores, it will split up the work and leverage all the available hardware to process the computation.

The Dask DataFrame architecture is what allows for this parallelism. On a 96-core machine, Dask can split up the data into 96 partitions, and process each partition at the same time.

Dask can also scale out a computation to run on multiple machines. Suppose you have a cluster of 3 computers, each with 24 cores. Dask can split up the data on multiple different machines and run the computations on all of the 72 available cores in the cluster.

Scaling up is scaling a workflow from using a single core to all the cores of a given machine. Scaling out is scaling a workflow to run on multiple computers in a cluster. Dask allows you to scale up or scale out, both of which are useful in different scenarios.

Summary

This chapter explained how Dask DataFrames are architected to overcome the scaling and big data performance issues of pandas. A good foundational understanding of Dask DataFrames will help you harness their power effectively.

Dask DataFrames add additional overhead, so they're not always faster than pandas. For small datasets, loading data into memory and running computations is fast and pandas performs quite well. Dask gains a performance advantage as dataset sizes grow and more powerful machines are used. The larger the data and the more cores a computer has, the more parallelism helps.

Dask is obviously the better option when the dataset is bigger than memory. pandas cannot work with datasets that are bigger than memory. Dask can run queries on datasets much larger than memory, as illustrated in our example.

Chapter 4 continues by digging more into Dask DataFrames. Chapter 4 will show you more important operations you can perform to manipulate your data. You're in a great place to understand the different types of Dask DataFrame operations now that you're familiar with how Dask DataFrames are architected.

Chapter 2. How to Work with Dask DataFrames

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

In the previous chapter, we explained the architecture of Dask DataFrames and how they're built on pandas DataFrames. We saw how pandas cannot scale to larger-than-memory datasets and how Dask overcomes this scaling limitation. Now it's time to dive into the specific tactics we'll need to master when working with Dask DataFrames.

In this chapter, we will apply the lessons we've learned to process large tabular datasets with Dask. We will walk through hands-on code examples to read, inspect, process, and write large datasets using Dask DataFrames. By working through these examples we will learn the Dask DataFrame API in depth and build understanding that will enable us to implement best practices when using Dask in real-world projects. It's a sort of 'behind the scenes' deep-dive into the end-to-end example Dask DataFrame we worked through in Chapter 2.

An important lesson to carry with us from the previous chapter is that pandas only uses one core to process computations while Dask can speed

up query execution time with parallel processing on multiple cores. Using Dask therefore means entering the world of *parallel computing*.

This means that even though much of the Dask DataFrame API syntax may appear familiar to pandas users, there are important underlying architectural differences to be aware of. The previous chapter explains those differences at a conceptual level. This chapter proceeds from there to dive into the hands-on application of the Dask API. It will explain how to leverage the power of parallel computing in practice.

As we proceed through this chapter, remember that:

- Dask DataFrames consist of **multiple partitions**, each of which is a pandas DataFrame.
- These partitions are **processed in parallel** by multiple cores at once.
- Dask uses **lazy evaluation** to optimize large-scale computations for parallel computation.

These fundamental concepts will be important to understand how to optimally leverage the Dask DataFrame API functions.

Reading Data into a Dask DataFrame

Imagine that you're working for a company that processes large-scale time series datasets for its customers. You are working on the Data Analysis team and have just received a copy of the file `0000.csv` from your colleague Azeem with the request to analyze patterns in the data. This sample file contains 1 week worth of data and is only 90 MB large, which means you can use pandas to analyze this subset just fine:

```
import pandas as pd
df = pd.read_csv("0000.csv")
```

Once the file has been loaded into a DataFrame, you can use pandas to analyze the data it contains. For example, use the head() method to see the first few rows of data in the DataFrame:

```
>>> df.head()
                      id     name         x         y
timestamp
1990-01-01 00:00:00  1066    Edith -0.789146  0.742478
1990-01-01 00:00:01   988   Yvonne  0.520779 -0.301681
1990-01-01 00:00:02  1022      Dan  0.523654 -0.438432
1990-01-01 00:00:03   944      Bob -0.768837  0.537302
1990-01-01 00:00:04   942   Hannah  0.990359  0.477812
...                   ...      ...       ...       ...
1990-01-03 23:59:55   927    Jerry -0.116351  0.456426
1990-01-03 23:59:56  1002    Laura -0.870446  0.962673
1990-01-03 23:59:57  1005  Michael -0.481907  0.015189
1990-01-03 23:59:58   975   Ingrid -0.468270  0.406451
1990-01-03 23:59:59  1059      Ray -0.739538  0.798155
```

Or calculate the number of records with positive values for a certain column:

```
>>> len(df[df.x > 0])

302316
```

Now that you've confirmed you can run analyses on the data, Azeem asks you to crunch the data for the first 6 months of the year. He has downloaded the data from the server as a single CSV which is about 2.4GB in size. You may still be able to process this with pandas, but it's pushing your machine to its limits and your analyses are running slow. This is a problem because Azeem has a deadline coming up and needs the results fast.

Fortunately, Dask can chop this single large 2.4GB file into smaller chunks and process these chunks in parallel. While pandas would have to process all the data on a single core, Dask can spread the computation out over all the cores in your machine. This will be much faster.

Read a single file into a Dask DataFrame

You can read a single CSV file like Azeem's 2.4GB `6M.csv` into a Dask DataFrame using the following syntax:

```
import dask.dataframe as dd
```

```
ddf = dd.read_csv("6M.csv")
```

> **NOTE**
>
> [NOTE] Note: we use `ddf` to refer to **D**ask **D**ata**F**rames and the conventional `df` to refer to pandas DataFrames.

Dask will read the data in this large single CSV file into a Dask DataFrame. A Dask DataFrame is always subdivided into appropriately sized ‘chunks’ called **partitions**. By cutting up the data into these conveniently smaller chunks, Dask DataFrame can distribute computations over multiple cores. This is what allows Dask to scale to much larger datasets than pandas. See the Working with Partitioned Data section for more details.

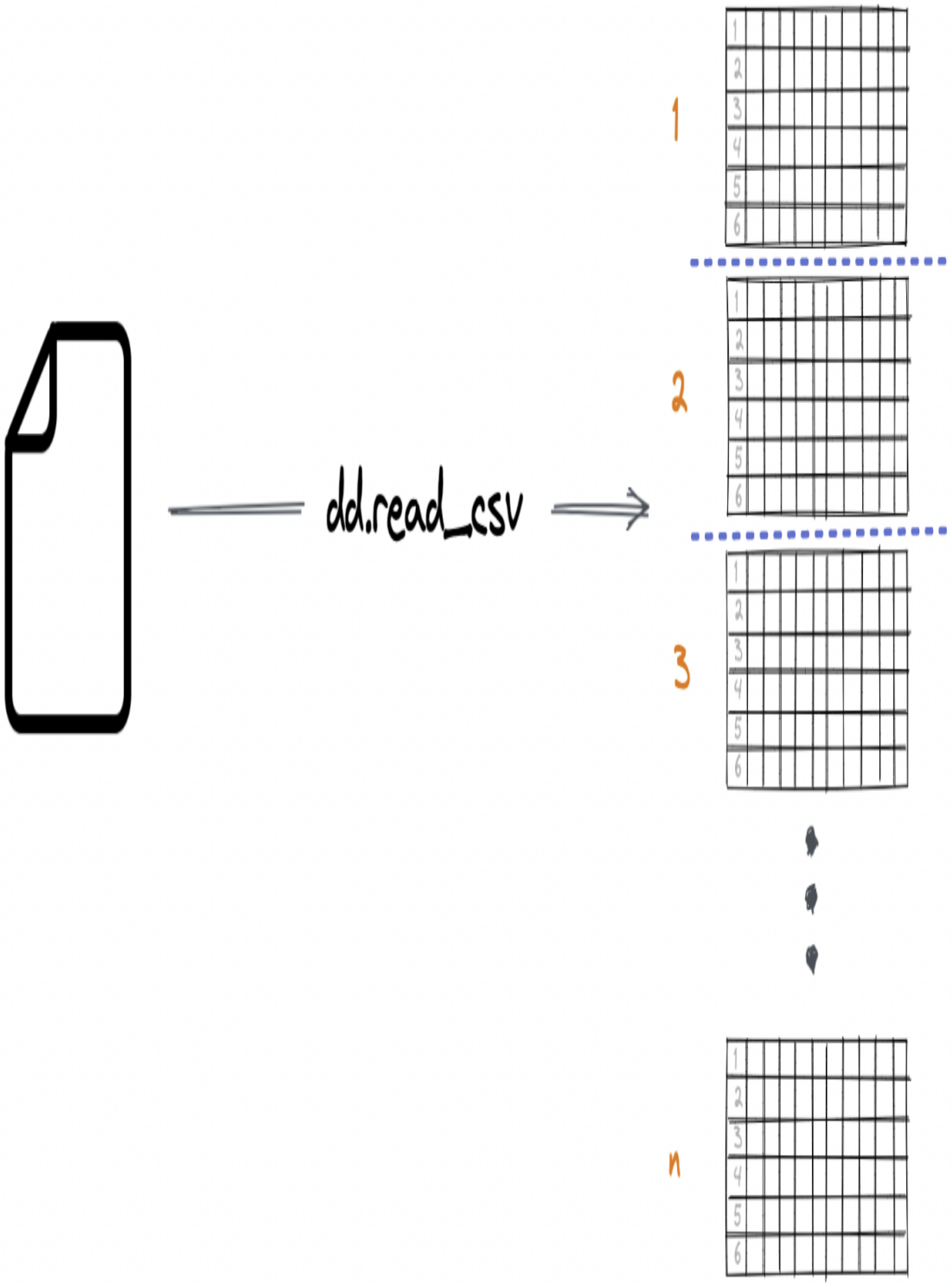

Figure 2-1. Dask reads a CSV file into a partitioned DataFrame.

Once your 2.4GB CSV file is loaded in, you can use Dask DataFrame to analyze the data it contains. This will look and feel much the same as it did with pandas, except you are no longer limited by the memory capacities of your machine and are able to run analyses on millions rather than thousands of rows. And perhaps most importantly, you are able to do so *before* Azeem's important deadline.

Read multiple files into a Dask DataFrame

After seeing you crunch the `6M.csv` file with the data for Q1 and Q2 successfully, Azeem is starting to see the power of Dask. He's now asked you to take a look at ***all the data*** he has available. That's twenty years' worth of time series data, totalling almost 60GB, scattered across 1,095 separate CSV files. This would have been an impossible task with pandas, but Dask has opened up a whole new world of possibilities for you here.

You ask Azeem to point you to the directory that contains the rest of the data, which looks like this:

```
$  ls
0000.csv
0001.csv
0002.csv
…
1094.csv
```

Dask DataFrame let's you use * as a glob string to read all of these separate CSV files into a Dask DataFrame at once:

```
ddf = dd.read_csv("*.csv")
```

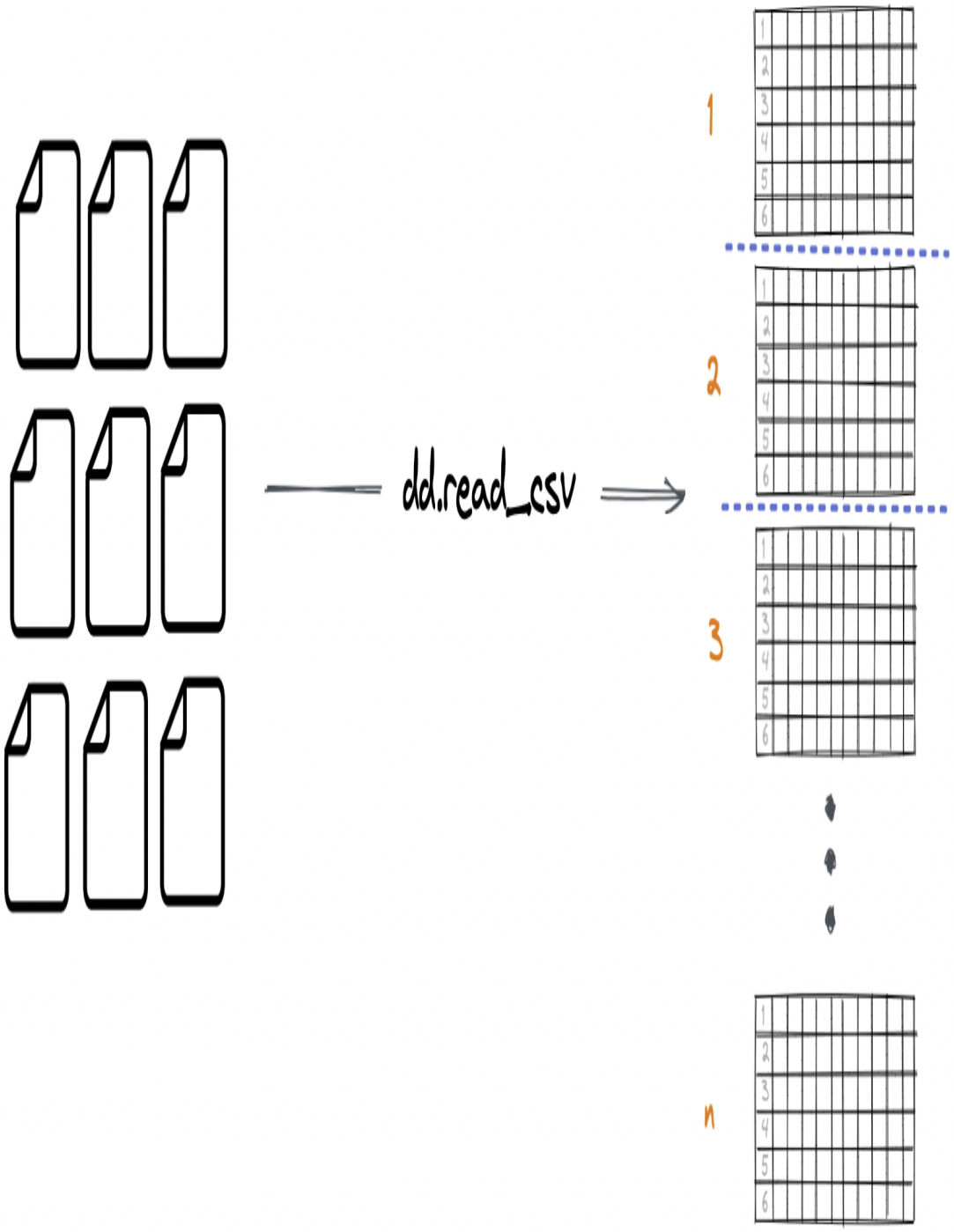
dd.read_csv
1
2
3
n

Figure 2-2. Dask reads multiple CSV files into a partitioned DataFrame.

Just like we saw above with the single CSV file, Dask will read the data across all of these separate CSV files into a single Dask DataFrame. The DataFrame will be divided into appropriately sized 'chunks' called **partitions**. Let's look more closely at how to work with Dask partitions.

Working with partitioned data

Dask intentionally and automatically splits up the data so operations can be performed on partitions of the data in parallel. This is done regardless of the original file format. It's important to understand how you can influence the partitioning of your data in order to choose a partitioning strategy that will deliver optimal performance for your dataset.

You can run `ddf.partitions` to see how many partitions the data is divided amongst.

```
>> ddf.partitions
1095
```

Setting Partition Size

By default, Dask will split the data into a certain number of partitions by calculating the optimal `blocksize`, which is based on the available memory and number of cores on the machine. This means the number of partitions for the same dataset may vary when working on different machines. Dask will ensure that the partitions are small enough to be processed quickly but not so small as to create unnecessary overhead. The maximum `blocksize` that Dask will calculate by default is 64 MB.

Dask splits up the data from the CSV files into one partition per file.

Reading Data into a Dask DataFrame

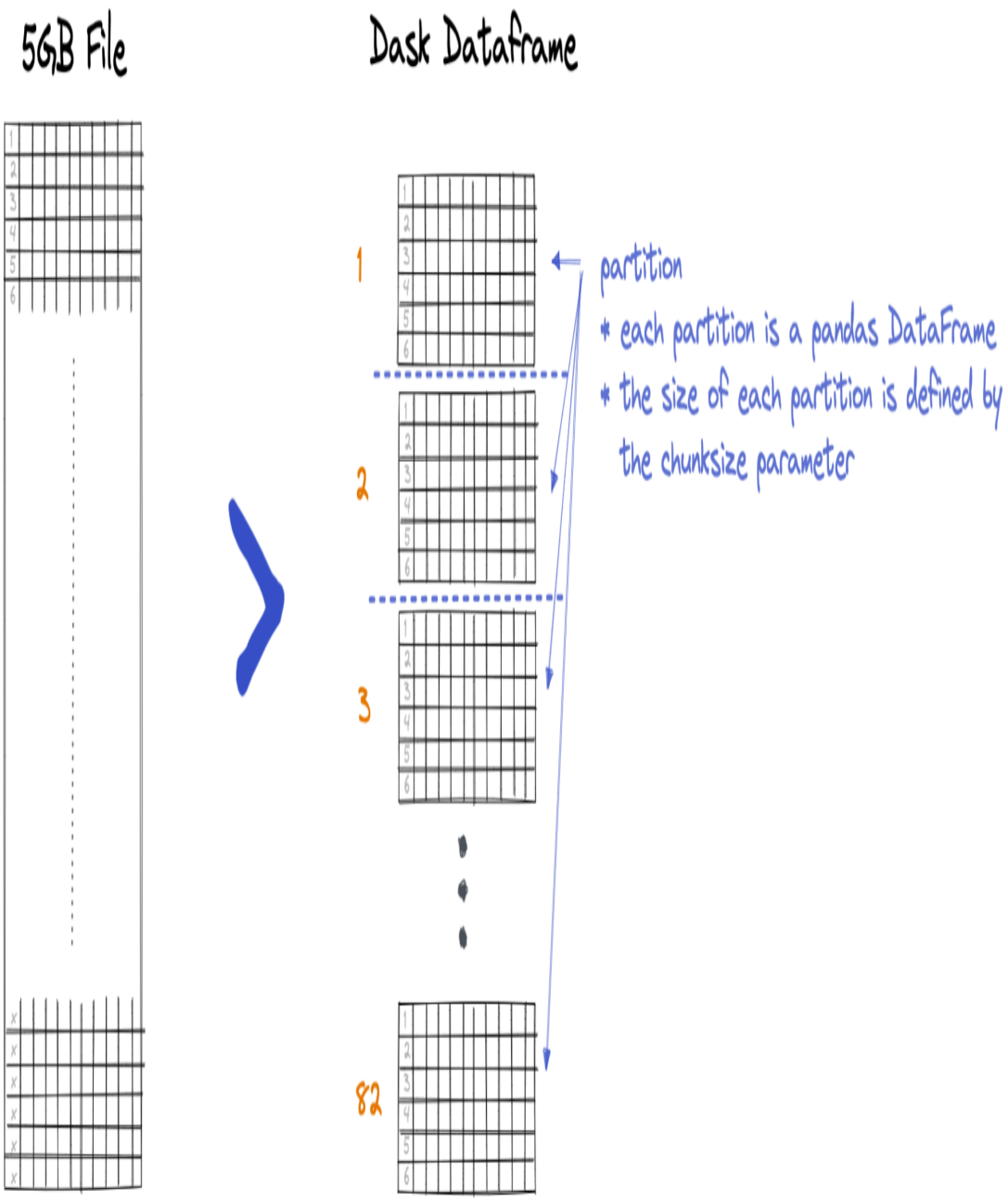

Figure 2-3. Dask reads a single large CSV file into multiple partitions.

You can manually set the number of partitions that the DataFrame contains using the `blocksize` parameter. You can tweak the partition size for optimal performance.

```
>>> ddf = dd.read_csv("*.csv", blocksize="16MB")
>>> ddf.npartitions
2190
```

This dataset will be read into a Dask DataFrame with 2190 partitions when the blocksize is set to 16 MB. The number of partitions goes up when the blocksize decreases.

Reading Data into a Dask DataFrame

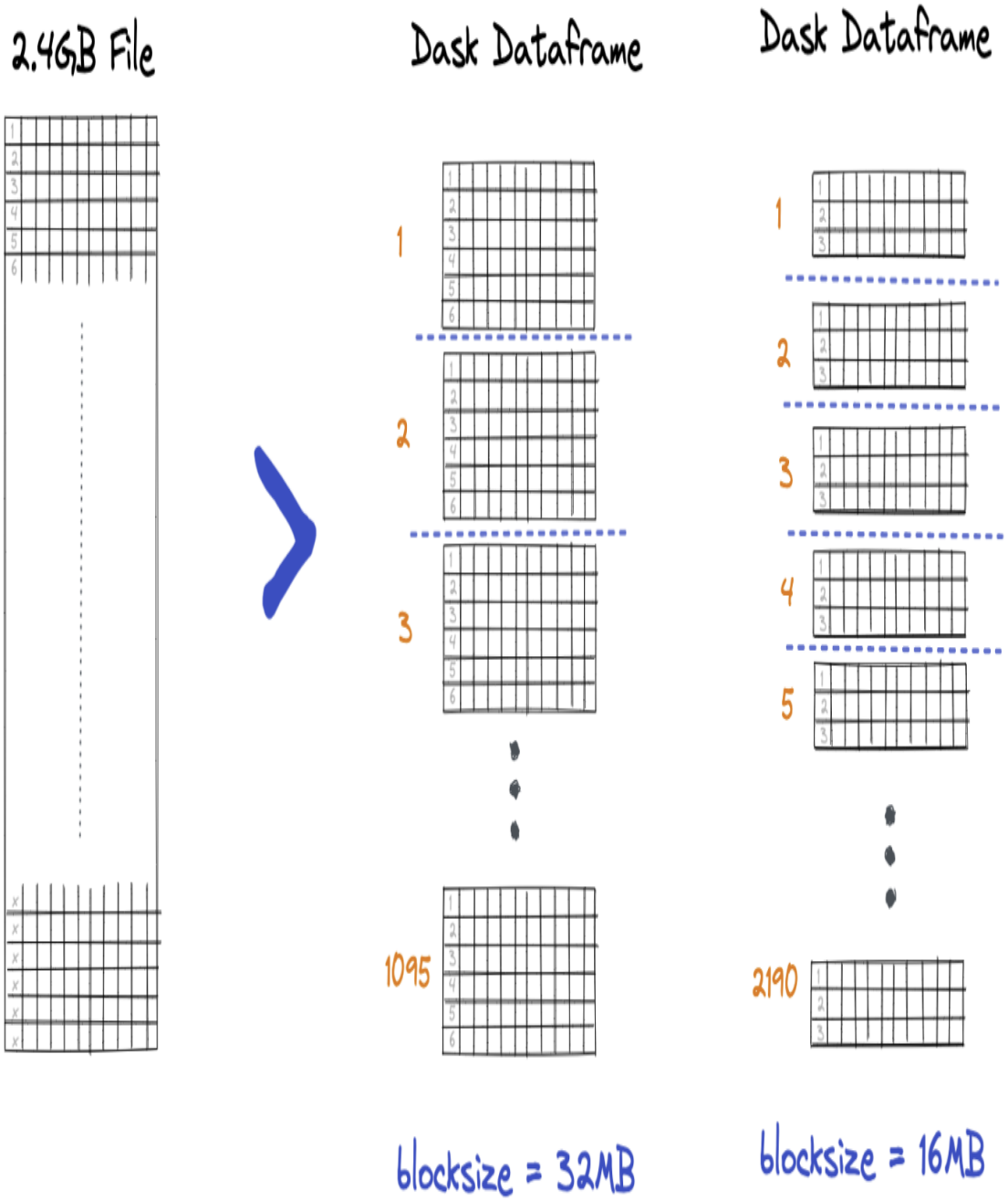

Figure 2-4. The number of partitions increases as the partitions become smaller.

> **NOTE**
>
> [TIP] As a rule of thumb, we recommend working with partition sizes of 100MB or less. This will ensure that the partitions are small enough to avoid bottlenecks but not so small that they start to incur unnecessary overhead.

Inspecting Data Types

After successfully reading the 1,095 CSV files into a single Dask DataFrame, you're now curious to know if the data for the single `0000.csv` file that was analyzed with pandas contains the same data types as the other 1,094 files in the folder. Remember that `df` is a pandas DataFrame containing the 93MB sample data and `ddf` is a Dask DataFrame containing 58 GB of data.

You can inspect the data types with pandas using the `dtypes` method:

```
        >> df.dtypes
timestamp       object
id               int64
name            object
x              float64
y              float64
dtype: object
```

pandas scans all the data to get the data types. This is fine for a 93MB file that easily fits into local memory. However, when working with Dask DataFrames we want to avoid reading all the data into memory. Loading all the CSV files into memory to get the data types will cause a `MemoryError`.

Instead, Dask *infers* the column data types (provided they are not explicitly set by the user). It does so by reading the first *n* rows of the DataFrame into memory and using the contents to infer the data types for the rest of the rows. *n* here is determined by the value of either the `sample` or

`sample_rows` arguments to `read_csv. sample` defines the number of bytes to read, `sample_rows` defines the number of rows to read.

Let's inspect the data types of our Dask DataFrame:

```
ddf = dd.read_csv("*.csv")
ddf.dtypes
timestamp       object
id              int64
name            object
x               float64
y               float64
dtype: object
```

> **NOTE**
>
> [CAUTION] You should be aware that inferring data types based on a sample of the rows is error-prone. Dask may incorrectly infer data types based on a sample of the rows which will cause downstream computations to error out.
>
> For example, the first 1000 rows of a column `id` may contain only integers and the 1001th row a string. If Dask is reading only the first 1000 rows, the dtype for `id` will be inferred as `int64`. This will lead to errors downstream when trying to run operations meant for integers on the column, since it contains at least one string. The correct dtype for this column should be `object`.

You can avoid data type inference by explicitly specifying dtypes when reading CSV files.

Let's manually set the name column to be a string, which is more efficient than object type columns:

```
>>> ddf = dd.read_csv(
    "*.csv",
    dtype={
        "name": "string[pyarrow]",
    },
)
>>> ddf.dtypes
timestamp     object
id             int64
name          string
```

```
x            float64
y            float64
dtype: object
```

Dask will infer the data types for the columns that you don't manually specify. If you specify the data types for all the columns, then Dask won't do any data type inference.

Reading Remote Data from the Cloud

After showcasing the power of Dask to your Team Lead using the entire CSV dataset, you've now been tasked with running analyses on all the data in production. Now it turns out that Azeem had actually downloaded the CSV files from the company's S3 bucket. That won't fly when running these crucial analyses in production and also happens to be against the company's data management best practices. You've been tasked with redoing the analysis without downloading the data locally.

Dask readers make it easy to read data that's stored in remote object data stores, like AWS S3.

Here's how to read a CSV file that's stored in a public S3 bucket to your local machine:

```
ddf = dd.read_csv(
    "s3://coiled-datasets/timeseries/20-years/csv/*.part"
)
```

> **NOTE**
>
> [TIP] Dask DataFrame readers expose a `storage_options` keyword that allows you to pass additional configurations to the remote data store, such as credentials for reading from private buckets or whether to use a SSL-encrypted connection. This can be helpful when working with private buckets or when your IT department has particular security policies in place.

Because of Dask's lazy evaluation, running the command above will complete quickly. But for subsequent operations, remember that the data

will actually need to be downloaded from the remote cloud storage to your local machine where the computations are running. This is called ‘moving data to compute’ and is generally inefficient.

Because of this, it’s usually best to run computations on remote data using cloud computing.

Here’s how to spin up a Dask cluster with Coiled and run these computations in the cloud. You will need a Coiled account to spin up these resources. If you’ve purchased this book, you get XXX free Coiled credits. Go to LINK to sign up for your personal Coiled account.

We’ll import Dask and Coiled:

```
import coiled
import dask
```

And then launch a Dask cluster with 5 workers:

```
cluster = coiled.Cluster(
    name="demo-cluster",
    n_workers=5
)
```

Finally, we’ll connect our local Dask client to the remote cluster:

```
client = dask.distributed.Client(cluster)
```

All subsequent Dask computations will now be executed in the cloud, instead of on your local machine.

Processing Data with Dask DataFrames

In the previous section we saw how to load data from various sources and file formats into a Dask DataFrame and highlighted considerations regarding partitioning and data type inference. You’ve now got your CSV data loaded into your Dask DataFrame and are ready to dig into the data.

Dask is intentionally designed to seamlessly scale existing popular PyData libraries like NumPy, scikit-learn and pandas. This means that processing data with Dask DataFrames will look and feel much like it would in pandas.

For example, the code below demonstrates how you can use Dask DataFrames to filter rows, compute reductions and perform groupby aggregations:

```
# filter rows
ddf[ddf.x > 0]
# compute max
ddf.x.max()
# perform groupby aggregation
ddf.groupby("id").x.mean()
```

However, there are a few cases where Dask operations require additional considerations, mainly because we are now operating in a parallel computing environment. The following section digs into those particular cases.

Converting to Parquet files

Azeem heard that Parquet files are faster and more efficient to query than CSV files. He's not sure about the performance benefits offered quite yet, but figures out how to make the conversion relatively easily with the help of a coworker:

```
ddf.to_parquet("s3://coiled-datasets/timeseries/20-years/parquet",
engine="pyarrow")
```

See the Benefits of Parquet section for more details.

Let's take a look at the various ways Azeem can query and manipulate the data in the 20-years Parquet dataset.

Materializing results in memory with compute

You can convert a Dask DataFrame to a pandas DataFrame with `compute()`. When the dataset is small, it's fine to convert to a pandas

DataFrame. In general, you don't want to convert Dask DataFrames to pandas DataFrames because then you lose all the parallelism and lazy execution benefits of Dask.

We saw earlier in the Chapter that Dask DataFrames are composed of a collection of underlying pandas DataFrames (partitions). Calling `compute()` concatenates all the Dask DataFrame partitions into a single Pandas DataFrame.

When the Dask DataFrame contains data that's split across multiple nodes in a cluster, then `compute()` may run slowly. It can also cause out of memory errors if the data isn't small enough to fit in the memory of a single machine.

Dask was created to solve the memory issues of using pandas on a single machine. When you run `compute()`, you'll face all the normal memory limitations of pandas.

Let's look at some examples and see when it's best to use compute() in your analyses.

Small DataFrame example

`compute()` converts a Dask DataFrame to a pandas DataFrame. Let's demonstrate with a small example.

Create a two column Dask DataFrame and then convert it to a pandas DataFrame:

```
>>> import dask.dataframe as dd
>>> import pandas as pd
>>> df = pd.DataFrame({
"col1": ["a", "b", "c", "d"],
"col2": [1, 2, 3, 4]
    })
>>> ddf = dd.from_pandas(df, npartitions=2)
>>> ddf.compute()
  col1  col2
0    a     1
1    b     2
```

```
2    c     3
3    d     4
```

Verify that `compute()` returns a pandas DataFrame:

```
>>> type(ddf.compute())
pandas.core.frame.DataFrame
```

Dask DataFrames are composed of many underlying pandas DataFrames, each of which is called a partition. It's no problem calling `compute()` when the data is small enough to get collected in a single pandas DataFrame, but this will break down whenever the data is too big to fit in the memory of a single machine.

Let's run `compute()` on a large DataFrame in a cloud cluster environment and take a look at the error message.

Large DataFrame example

Create a 5 node Dask cluster and read in a 662 million row dataset into a Dask DataFrame:

```
cluster = coiled.Cluster(name="demo-cluster", n_workers=5)
client = dask.distributed.Client(cluster)
ddf = dd.read_parquet(
    "s3://coiled-datasets/timeseries/20-years/parquet",
    storage_options={"anon": True, "use_ssl": True},
    engine="pyarrow",
)
```

Perform a filtering operation on the Dask DataFrame and then collect the result into a single pandas DataFrame with `compute()`:

```
res = ddf.loc[ddf["id"] > 1150]
res.compute()
```

This works because `res` only has 1,103 rows of data. It's easy to collect such a small dataset into a single pandas DataFrame.

`ddf.compute()` will error out if we try to collect the entire 663 million row dataset into a pandas DataFrame.

Azeem's dataset has 58 GB of data, which is too large for a single machine.

Compute intuition

Here's a diagram that visually demonstrates how compute() works, to give you some additional intuition.

Suppose you have a 3-node cluster with 4 partitions. You run a compute() operation to collect all of the data in a single Pandas DataFrame.

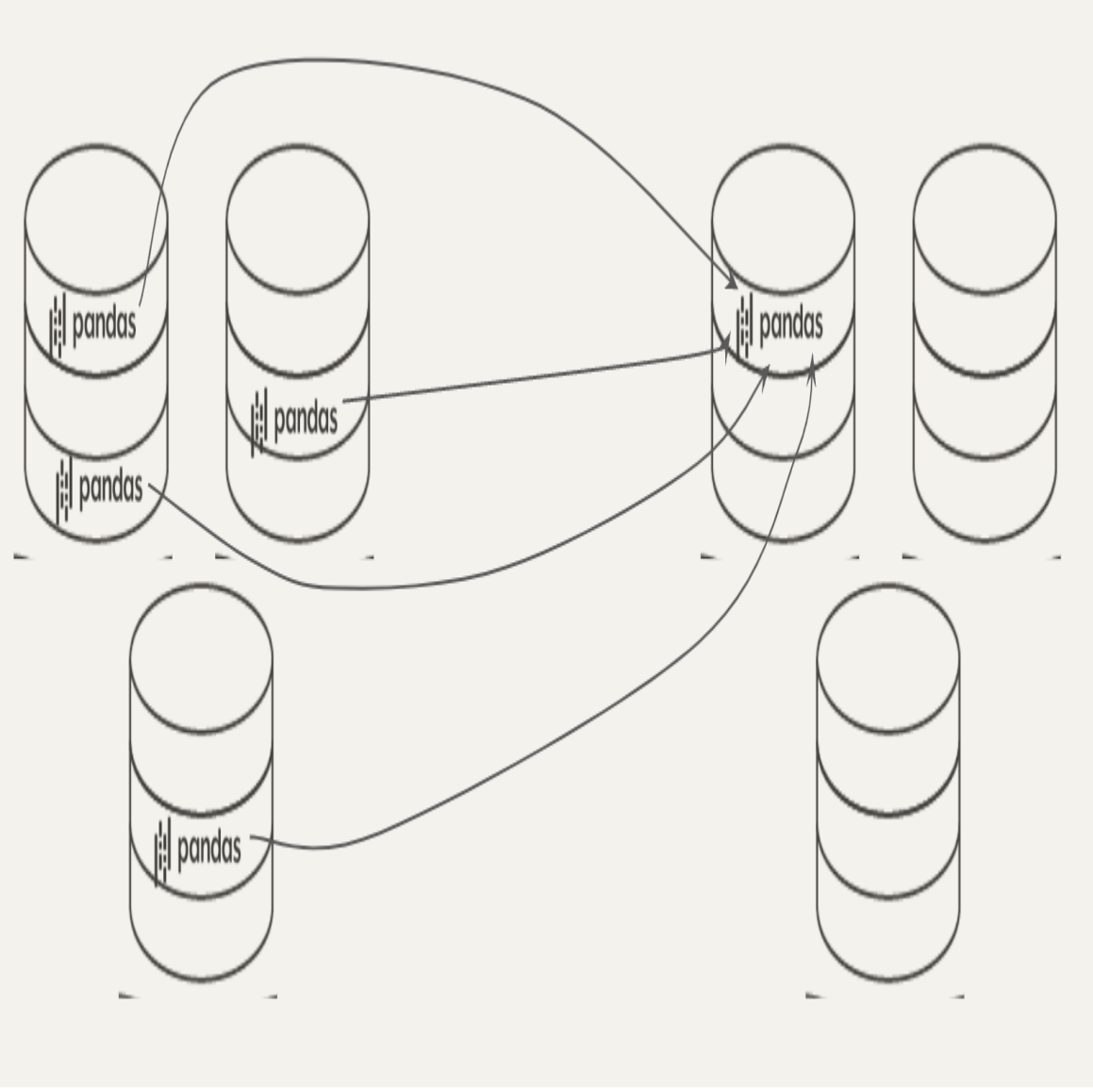

Figure 2-5. Calling compute() may create memory errors.

This diagram clearly illustrates why the compute() operation can cause out of memory exceptions. Data that fits when it's split across multiple machines in a cluster won't necessarily fit in a single pandas DataFrame on only one machine.

Minimizing compute() calls

compute() can be an expensive operation, so you want to minimize compute() calls whenever possible.

The following code snippet runs compute() twice and takes 63 seconds to run on a 5 node Dask cluster.

```
%%time
id_min = ddf.id.min().compute()
id_max = ddf.id.max().compute()
CPU times: user 442 ms, sys: 31.4 ms, total: 473 ms
Wall time: 1min 20s
```

We can refactor this code to only run compute() once and it'll run in 33 seconds.

```
%%time
id_min, id_max = dask.compute(ddf.id.min(), ddf.id.max())
CPU times: user 222 ms, sys: 19.1 ms, total: 241 ms
Wall time: 35.5 s
```

Fewer compute() calls will always run faster than more compute() invocations because Dask can optimize computations with shared tasks.

When to call compute()

You can call compute() if you've performed a large filtering operation or another operation that decreases the size of the overall dataset. If your data comfortably fits in memory, you don't need to use Dask. Just stick with pandas if your data is small enough.

You will also call compute() when you'd like to force Dask to execute the computations and return a result. Dask executes computations lazily by

default. It'll avoid running expensive computations until you run a method like compute() that forces computations to be executed.

Materializing results in memory with persist()

Instead of calling compute(), you can store Dask DataFrames in memory with persist(). This will store the contents of the Dask DataFrame in cluster memory which will make downstream queries that depend on the persisted data faster. This is great when you perform some expensive computations and want to save the results in memory so they're not rerun multiple times.

> **NOTE**
>
> [CAUTION] Be careful with using persist() when working locally. Calling persist() will load the results of a computation into memory, since you are not operating with a cluster to give you additional resources, this may mean that persist() creates memory errors. Using persist() is most beneficial when working with a remote cluster.

> **NOTE**
>
> [NOTE] Note that persist() commands do not block. This means that you can continue running other commands immediately afterwards and the persist() computation will run in the background.

Many Dask users erroneously assume that Dask DataFrames are persisted in memory by default, which isn't true. Dask runs computations in memory. It doesn't store data in memory unless you explicitly call `persist()`.

Let's start with examples of persist() on small DataFrames and then move to examples on larger DataFrames so you can see some realistic performance benchmarks.

Simple example

Let's create a small Dask DataFrame to demonstrate how persist() works:

```
>>> import dask.dataframe as dd
>>> import pandas as pd
>>> df = pd.DataFrame({
        "col1": ["a", "b", "c", "d"],
   "col2": [1, 2, 3, 4]
    })
>>> ddf = dd.from_pandas(df, npartitions=2)
>>> persisted_ddf = ddf.persist()
>>> len(persisted_ddf)
4
```

The persisted_ddf is saved in memory when `persist()` is called. Subsequent queries that run off of `persisted_ddf` will execute more quickly than if you hadn't called persist().

Let's run `persist()` on a larger DataFrame to see some real computation runtimes.

Example on big dataset

Let's run some queries on a dataset that's not persisted to get some baseline query runtimes. Then let's persist the dataset, run the same queries, and quantify the performance gains from persisting the dataset.

These queries are run on Azeem's 20 year Parquet timeseries dataset.

Here's the code that creates a computation cluster, reads in a DataFrame, and then creates a filtered DataFrame.

```
import coiled
import dask
import dask.dataframe as dd
cluster = coiled.Cluster(name="powers", n_workers=5)

client = dask.distributed.Client(cluster)
ddf = dd.read_parquet(
    "s3://coiled-datasets/timeseries/20-years/parquet",
    storage_options={"anon": True, "use_ssl": True},
    engine="pyarrow",
)
```

Let's time a couple of analytical queries.

```
>>> res = ddf.loc[ddf["id"] > 1150]
>>> len(res)
87 seconds
>>> res.name.nunique().compute()
62 seconds
```

Let's persist the dataset to cluster memory and then run the same queries to see how long they take to execute.

```
>>> persisted_res = res.persist()
>>> len(persisted_res)
2 seconds
>>> persisted_res.name.nunique().compute()
2 seconds
```

The queries took over a minute to run before the data was persisted, but only takes 2 seconds to run on the persisted dataset.

Of course it takes some time to persist the data. Let's look at why this particular example gave us great results when persisting.

NOTE

[NOTE] Great opportunity to persist

Persisting helps sometimes and causes problems other times. Let's look at high level patterns when it'll usually help and when it'll usually cause problems.

Our example uses the following pattern:

- Start with a large dataset
- Filter it down to a much smaller datasets (that's much smaller than the memory of the cluster)
- Run analytical queries on the filtered dataset

You can expect good results from persist() with this set of circumstances.

Here's a different pattern that won't usually give such a good result.

- Read in a large dataset that's bigger than memory
- Persist the entire dataset
- Run a single analytical operation

In this case, the cost of running the persist operation will be greater than the benefits of having a single query run a little bit faster. Persisting doesn't always help.

Writing to disk vs persisting in memory

We can also "persist" results by writing to disk rather than saving the data in memory.

Let's persist the filtered dataset in S3 and run the analytical queries to quantify time savings.

```
>>> res.repartition(2).to_parquet(
        "s3://coiled-datasets/tmp/matt/disk-persist",
        engine="pyarrow"
    )
>>>> df = dd.read_parquet(
        "s3://coiled-datasets/tmp/matt/disk-persist",
        storage_options={"anon": True, "use_ssl": True},
        engine="pyarrow",
    )
>>> len(df)
```

```
0.4 seconds
>>> df.name.nunique().compute()
0.3 seconds
```

The filtered dataset that was written to disk can be queried with subsecond response times.

Writing temporary files to disk isn't always ideal because then you have stale files sitting around that need to get cleaned up later.

Repartitioning and persisting

We can also repartition before persisting, which will make our analytical queries in this example run even faster.

```
>>> res2 = res.repartition(2)
>>> persisted_res2 = res2.persist()
>>> len(persisted_res2)
0.3 seconds
>>> persisted_res2.name.nunique().compute()
0.3 seconds
```

The filtered dataset is tiny and doesn't need a lot of partitions. That's why repartitioning drops query times from around 2 seconds to 0.3 seconds in this example.

COMPUTE VS PERSIST

Compute and persist are both ways to materialize results in memory.

Compute materializes results in a pandas DataFrame and persist materializes the results in a Dask DataFrame.

Compute only works for datasets that fit in the memory of a single machine. Larger results can be persisted because the data can be spread in the memory of multiple computers in a cluster.

Persist summary

Persist is a powerful optimization technique to have in your Dask toolkit.

It's especially useful when you've performed some expensive operations that reduce the dataset size and subsequent operations benefit from having the computations stored.

Some new Dask programmers can misuse persist() and slow down analyses by persisting too often or trying to persist massive datasets. It helps sometimes, but it can cause analyses to run slower when used incorrectly.

Persisting will generally speed up analyses when one or more items in this set of facts are true:

- You've performed expensive computations that have reduced the dataset size
- The reduced dataset comfortably fits in memory
- You want to run multiple queries on the reduced dataset

Repartitioning Dask DataFrames

This section explains how to redistribute data among partitions in a Dask DataFrame with repartitioning. Analyses run slower when data is unevenly distributed across partitions, and repartitioning can smooth out the data and provide significant performance boost.

Dask DataFrames consist of partitions, each of which is a pandas DataFrame. Dask performance will suffer if there are lots of partitions that are too small or some partitions that are too big. Repartitioning a Dask DataFrame solves the issue of "partition imbalance".

Let's start with some simple examples to get you familiar with the repartition() syntax.

Simple examples

Let's create a Dask DataFrame with six rows of data organized in three partitions:

```
import pandas as pd
```

```
import dask.dataframe as dd
df = pd.DataFrame({
    "nums":[1, 2, 3, 4, 5, 6],
    "letters":["a", "b", "c", "d", "e", "f"]
})
ddf = dd.from_pandas(df, npartitions=3)
```

Print the content of each DataFrame partition:

```
>>>> for i in range(ddf.npartitions):
        print(ddf.partitions[i].compute())
  nums letters
0     1       a
1     2       b
   nums letters
2     3       c
3     4       d
   nums letters
4     5       e
5     6       f
```

Repartition the DataFrame into two partitions:

```
>>> ddf2 = ddf.repartition(2)
>>> for i in range(ddf2.npartitions):
        print(ddf2.partitions[i].compute())
  nums letters
0     1       a
1     2       b
   nums letters
2     3       c
3     4       d
4     5       e
5     6       f
```

repartition(2) causes Dask to combine partition 1 and partition 2 into a single partition. Dask's repartition algorithm is smart to coalesce existing partitions and avoid full data shuffles.

You can also increase the number of partitions with repartition. Repartition the DataFrame into 5 partitions:

```
>>> ddf5 = ddf.repartition(5)
>>> for i in range(ddf5.npartitions):
```

```
        print(ddf5.partitions[i].compute())
  nums letters
0     1       a
   nums letters
1     2       b
   nums letters
2     3       c
   nums letters
3     4       d
   nums letters
4     5       e
5     6       f
```

In practice, it's easier to repartition by specifying a target size for each partition (e.g. 100 MB per partition). You want Dask to do the hard work of figuring out the optimal number of partitions for your dataset. Here's the syntax for repartitioning into 100MB partitions:

```
ddf.repartition(partition_size="100MB")
```

When to repartition

Of course, repartitioning isn't free and takes time. The cost of performing a full data shuffle can outweigh the benefits of subsequent query performance.

You shouldn't always repartition whenever a dataset is imbalanced. Repartitioning should be approached on a case-by-case basis and only performed when the benefits outweigh the costs.

Common causes of partition imbalance

Filtering is a common cause of DataFrame partition imbalance.

Suppose you have a DataFrame with a first_name column and the following data:

- Partition 0: Everyone has a first_name "Allie"
- Partition 1: Everyone has first_name "Matt"
- Partition 2: Everyone has first_name "Sam"

If you filter for all the rows with first_name equal to “Allie”, then Partition 1 and Partition 2 will be empty. Empty partitions cause inefficient Dask execution. It’s often wise to repartition after filtering.

Filtering Dask DataFrames

This section explains how to filter Dask DataFrames based on the DataFrame index and on column values using loc().

Filtering Dask DataFrames can cause data to be unbalanced across partitions which isn’t desirable from a performance perspective. This section illustrates how filtering can cause the “empty partition problem” and how to eliminate empty partitions with the repartitioning techniques we learned in the previous section.

Index filtering

Dask DataFrames consist of multiple partitions, each of which is a pandas DataFrame. Each pandas DataFrame has an index. Dask allows you to filter multiple pandas DataFrames on their index in parallel, which is quite fast.

Let’s create a Dask DataFrame with 6 rows of data organized in two partitions:

```
import pandas as pd
import dask.dataframe as dd
df = pd.DataFrame({
"nums":[1, 2, 3, 4, 5, 6],
"letters":["a", "b", "c", "d", "e", "f"]
})
ddf = dd.from_pandas(df, npartitions=2)
```

Let’s visualize the data in each partition:

```
>>> for i in range(ddf.npartitions):
        print(ddf.partitions[i].compute())
  nums letters
0     1       a
1     2       b
2     3       c
   nums letters
```

```
3     4       d
4     5       e
5     6       f
```

Dask automatically added an integer index column to our data.

Grab rows 2 and 5 from the DataFrame:

```
>>> ddf.loc[[2, 5]].compute()
  nums letters
2     3       c
5     6       f
```

Grab rows 3, 4, and 5 from the DataFrame:

```
>>> ddf.loc[3:5].compute()
  nums letters
3     4       d
4     5       e
5     6       f
```

Let's learn more about how Dask tracks information about divisions in sorted DataFrames to perform `loc` filtering efficiently.

Divisions refresher

We've already discussed Dask DataFrame divisions in the architecture chapter, but we're going to do a full refresher here because they're so critical when working with the Dask DataFrames index.

Dask is aware of the starting and ending index value for each partition in the DataFrame and stores this division's metadata to perform quick filtering.

You can verify that Dask is aware of the divisions for this particular DataFrame by running `ddf.known_divisions` and seeing it returns `True`. Dask isn't always aware of the DataFrame divisions.

Print all the divisions of the DataFrame:

```
>>> ddf.divisions
(0, 3, 5)
```

Take a look at the values in each partition of the DataFrame to better understand this divisions output.

```
>>> for i in range(ddf.npartitions):
        print(ddf.partitions[i].compute())
  nums letters
0     1       a
1     2       b
2     3       c
   nums letters
3     4       d
4     5       e
5     6       f
```

The first division is from 0-3, and the second division is from 3-5. This means the first division contains rows 0 to 2, and the last division contains rows 3 to 5.

Dask's division awareness in this example lets it know exactly what partitions it needs to fetch from when filtering.

Column value filtering

You won't always be able to filter based on index values. Sometimes you need to filter based on actual column values.

Fetch all rows in the DataFrame where `nums` is even:

```
>>> ddf.loc[ddf["nums"] % 2 == 0].compute()
  nums letters
1     2       b
3     4       d
5     6       f
```

Find all rows where `nums` is even, and `letters` contains either `b` or `f`:

```
>>> ddf.loc[(ddf["nums"] % 2 == 0) & (ddf["letters"].isin(["b",
"f"]))].compute()
  nums letters
1     2       b
5     6       f
```

Dask makes it easy to apply multiple logic conditions when filtering.

Empty partition problem

Let's read Azeem's 662 million rows Parquet dataset into a Dask DataFrame and perform a filtering operation to illustrate the empty partition problem.

Read in the data and create the Dask DataFrame:

```
ddf = dd.read_parquet(
    "s3://coiled-datasets/timeseries/20-years/parquet",
    storage_options={"anon": True, 'use_ssl': True}
)
```

`ddf.npartitions` shows that the DataFrame has 1,095 partitions:

```
>>> ddf.npartitions
1095
```

Here's how to filter the DataFrame to only include rows with an id greater than 1150:

```
res = ddf.loc[ddf["id"] > 1150]
```

Run `len(res)` to see that the DataFrame only has 1,103 rows after this filtering operation:

```
>>> len(res)
1103
```

This was a big filter, and only a small fraction of the original 662 million rows remain.

We can run `res.npartitions` to see that the DataFrame still has 1,095 partitions. The filtering operation didn't change the number of partitions:

```
>>> res.npartitions
1095
```

Run `res.map_partitions(len).compute()` to visually inspect how many rows of data are in each partition:

```
>>> res.map_partitions(len).compute()
0       0
1       1
2       0
3       0
4       2
       ..
1090    0
1091    2
1092    0
1093    0
1094    0
Length: 1095, dtype: int64
```

A lot of the partitions are empty and others only have a few rows of data.

Dask often works best with partition sizes of at least 100MB. Let's repartition our data to two partitions and persist it in memory:

```
res2 = res.repartition(2).persist()
```

Subsequent operations on res2 will be really fast because the data is stored in memory. `len(res)` takes 57 seconds whereas `len(res2)` only takes 0.3 seconds.

The filtered dataset is so small in this example that you could even convert it to a pandas DataFrame with `res3 = res.compute()`. It only takes 0.000011 seconds to execute `len(res3)`.

> **NOTE**
>
> [TIP] You don't have to filter datasets at the computation engine level. You can also filter at the database level and only send a fraction of the data to the computation engine.

Query pushdown

Query pushdown is when you perform data operations before sending the data to the Dask cluster. Part of the work is "pushed down" to the database level.

Here's the high level process for filtering with a Dask cluster:

- Read all the data from disk into the cluster
- Perform the filtering operation
- Repartition the filtered DataFrame
- Possibly write the result to disk (ETL style workflow) or persist in memory

Organizations often need to optimize data storage and leverage query pushdown in a manner that's optimized for their query patterns and latency needs.

Best practices

Dask makes it easy to filter DataFrames, but you need to be cognizant of the implications of big filters.

After filtering a lot of data, you should consider repartitioning and persisting the data in memory.

You should also consider filtering at the database level and bypassing cluster filtering altogether. Lots of Dask analyses run slower than they should because a large filtering operation was performed, and the analyst is running operations on a DataFrame with tons of empty partitions.

Setting the Index

Indexes are used by normal pandas DataFrames, Dask DataFrames, and many databases in general. Indexes let you efficiently find rows that have a certain value, without having to scan each row.

In plain pandas, it means that after a `set_index("col")`, `df.loc["foo"]` is faster than `df[df.col == "foo"]` was before.

The loc() uses an efficient data structure that only has to check a couple rows to figure out where “foo” is. Whereas `df[df.col == "foo"]` has to scan every single row to see which ones match.

The thing is, computers are very, very fast at scanning memory, so when you’re running pandas computations on one machine, index optimizations aren’t as important. But scanning memory across many distributed machines is not fast. So index optimizations that you don’t notice much with pandas makes an enormous difference with Dask.

Dask DataFrames Divisions

Remember that a Dask DataFrame is composed of many pandas DataFrames, potentially living on different machines, each one of which we call a partition. Each of these partitions is a pandas DataFrame that has its own index.

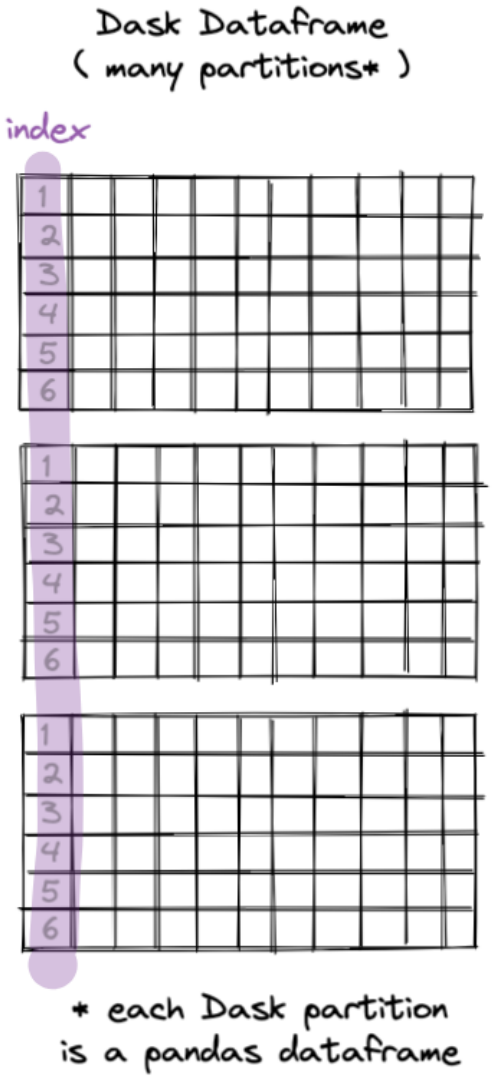

Figure 2-6. Figure title needed.

Dask DataFrame has its own version of an index for the distributed DataFrame as a whole, called `divisions`. The `divisions` are like an index for the indexes—it tracks the index bounds for each partition, so you can easily tell which partition contains a given value (just like pandas's index tracks which row will contain a given value).

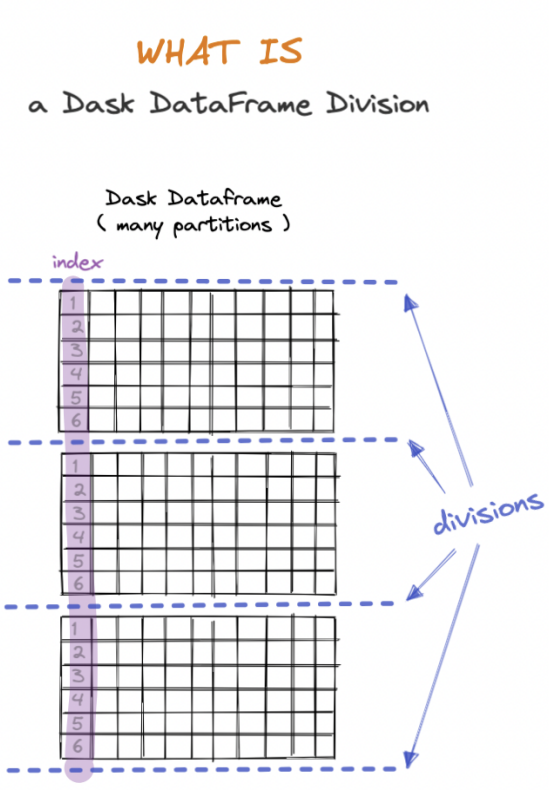

Figure 2-7. Figure title needed.

When there are millions of rows spread across hundreds of machines, it's too much to track every single row. We just want to get in the right ballpark—which machine will hold this value?—and then tell that machine to go find the row itself.

So `divisions` is just a simple list giving the lower and upper bounds of values that each partition contains. Using this, Dask does a quick binary search locally to figure out which partition contains a given value.

Just like with a pandas index, having known divisions lets us change a search that would scan every row (`df[df.col == "foo"]`) to one that quickly goes straight to the right place (`df.loc["foo"]`).

How to Set the Index

You can set the index of a Dask DataFrame using the `set_index()` method:

```
ddf2 = ddf.set_index("id")
```

Notice that `ddf.set_index("id")` does not specify divisions, so Dask needs to go figure them out. To do this, it has to load and compute all of `ddf` immediately to look at the distribution of its values. Then, when you later call `.compute()`, it'll load and compute `ddf` a second time. This is slow in general, and particularly bad if the DataFrame already has lots of operations applied to it—all those operations also have to run twice.

Instead, when divisions are passed, Dask doesn't need to compute the whole DataFrame to figure them out, which is obviously a lot faster. To pick good divisions, you must use your knowledge of the dataset. What range of values is there for the column? What sort of general distribution do they follow—a normal bell curve, a continuous distribution? Are there known outliers?

Another strategy is to let Dask compute the divisions once, then copy-paste them to reuse later:

```
dask_computed_divisions = ddf3.set_index("id").divisions
unique_divisions =
list(dict.fromkeys(list(dask_computed_divisions)))
print(repr(unique_divisions))
# ^ copy this and reuse
```

This is especially helpful if you'll be rerunning a script or notebook on the same (or similar) data many times. However, you shouldn't set divisions if the data you're processing is very unpredictable. In that case, it's better to spend the extra time and let Dask re-compute good divisions each time.

When to Set the Index

Since `set_index()` is an expensive operation, you should only run the computation when it'll help subsequent computations run faster.

Here are the main operations that'll run faster when an index is set:

- Filtering on the index

- Joining on the index
- Groupby on the index
- Sophisticated custom code in map_partitions (advanced use case)

By all means, set indexes whenever it'll make your analysis faster. Just don't run these expensive computations unnecessarily.

For example, you shouldn't always `set_index()` before you `.loc`. If you just need to pull a value out once, it's not worth the cost of a whole shuffle. But if you need to pull lots of values out, then it is. Same with a merge: if you're just merging a DataFrame to another, don't `set_index()` first (the merge will do this internally anyway). But if you're merging the same DataFrame multiple times, then the `set_index()` is worth it.

As a rule of thumb, you should `set_index()` if you'll do a merge, `groupby(df.index)`, or `.loc` on the re-indexed DataFrame more than once. You may also want to re-index your data before writing it to storage in a partitioned format like Parquet. That way, when you read the data later, it's already partitioned the way you want, and you don't have to re-index it every time.

Joining Dask DataFrames

In this section we'll look at how to merge Dask DataFrames and discuss important considerations when making large joins. We'll learn how to join a large Dask DataFrame to a small pandas DataFrame, how to join two large Dask DataFrames and how to structure our joins for optimal performance.

Join a Dask DataFrame to a pandas DataFrame

We can join a Dask DataFrame to a small pandas DataFrame by using the `dask.dataframe.merge()` method, similar to the pandas api. Below we execute a left join on our Dask DataFrame `ddf` with a small pandas DataFrame `df` that contains a boolean value for every name in the dataset:

```
>>> # load in small pandas dataframe
>>> df = pd.read_csv("small_df.csv", index_col=0)
>>> # merge dask dataframe to pandas dataframe
>>> join = ddf.merge(
        df,
        how="left",
        on=["name"]
    )
>>> # materialize first 5 results
>>> join.head()
```

	id	name	x	y	checked
0	1008	Dan	-0.259374	-0.118314	False
1	987	Patricia	0.069601	0.755351	True
2	980	Zelda	-0.281843	-0.510507	True
3	1020	Ursula	-0.569904	0.523132	True
4	967	Michael	-0.251460	0.810930	True

Figure 2-8. Figure title needed.

If you're working with a small Dask DataFrame instead of a pandas DataFrame, you have two options. You can convert it into a pandas DataFrame using compute(). This will load the DataFrame into memory. Alternatively, if you can't or don't want to load it into your single machine memory, you can turn the small Dask DataFrame into a single partition by using the repartition() method instead. These two operations are programmatically equivalent which means there's no meaningful difference

in performance between them. See the Compute and Repartitioning sections respectively for more details.

HOW TO

Join a Dask DataFrame to a pandas DataFrame

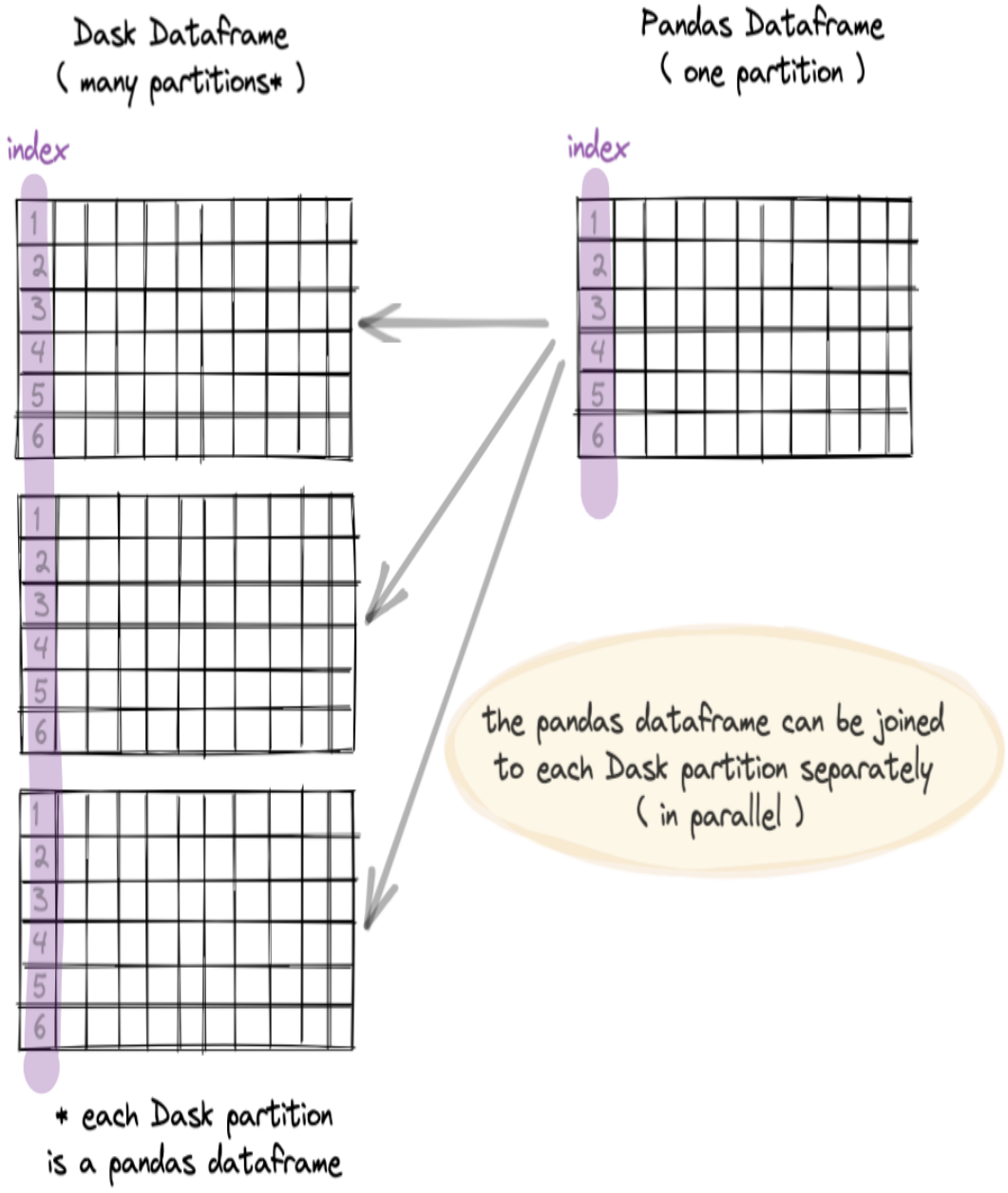

Figure 2-9. Figure title needed.

Joining two Dask DataFrames

To join two large Dask DataFrames, you can use the exact same syntax. In this case we are specifying the `left_index` and `right_index` keywords to tell Dask to use the indices of the two DataFrames as the columns to join on. This will join the data based on the timestamp column:

```
>>> large_join = ddf.merge(
        ddf_2,
        how="left",
        left_index=True,
        right_index=True
    )
>>> # materialize first 5 results
>>> large_join.head()
```

timestamp	id_x	name_x	x_x	y_x	id_y	name_y	x_y	y_y
2000-01-01 00:00:00	1008	Dan	-0.259374	-0.118314	1011	Xavier	-0.469186	0.812331
2000-01-01 00:00:01	987	Patricia	0.069601	0.755351	1015	Frank	0.261871	-0.854165
2000-01-01 00:00:02	980	Zelda	-0.281843	-0.510507	1034	Ursula	-0.002140	-0.467836
2000-01-01 00:00:03	1020	Ursula	-0.569904	0.523132	980	Yvonne	-0.740036	0.120821
2000-01-01 00:00:04	967	Michael	-0.251460	0.810930	1003	Michael	0.291326	-0.329360

Figure 2-10. Figure title needed.

However, merging two large Dask DataFrames requires careful consideration of your data structure and the final result you're interested in. Joins are expensive operations, especially in a distributed computing context. Understanding both your data and your desired end result can help you set up your computations efficiently to optimize performance. The most important consideration is whether and how to set your DataFrame's index before executing the join.

Considerations when joining Dask DataFrames

Joining two DataFrames can be either very expensive or very cheap depending on the situation. It is cheap in the following cases:

- Joining a Dask DataFrame with a Pandas DataFrame
- Joining a Dask DataFrame with another Dask DataFrame of a single partition
- Joining Dask DataFrames along their indexes

As explained earlier in the book, Dask DataFrames are divided into partitions, where each single partition is a pandas DataFrame. Dask can track how the data is partitioned (i.e. where one partition starts and the next begins) using a DataFrame's divisions. If a Dask DataFrame's divisions are known, then Dask knows the minimum value of every partition's index and the maximum value of the last partition's index. This enables Dask to take efficient shortcuts when looking up specific values. Instead of searching the entire dataset, it can find out which partition the value is in by looking at the divisions and then limit its search to only that specific partition. This is called a sorted join.

If divisions are not known, then Dask will need to move all of your data around so that rows with matching values in the joining columns end up in the same partition. This is called an unsorted join and it's an extremely memory-intensive process, especially if your machine runs out of memory and Dask will have to read and write data to disk instead. This is a situation you want to avoid.

If you are planning to run repeated joins against a large Dask DataFrame, it's best to sort the Dask DataFrame using the `set_index()` method first to improve performance. See Section 2.5 above for more on the `set_index()` method and `divisions`.

It's good practice to write sorted DataFrames to the Apache Parquet file format in order to preserve the index. See the Working with Parquet Section for more on the benefits of working with this data format.

Mapping Custom Functions

This section provides a quick refresher of how we can run custom functions on pandas DataFrames and then demonstrates how we can parallelize these operations on Dask DataFrames. Dask makes it easy to apply custom functions on each of the underlying pandas DataFrames it contains.

pandas apply refresher

pandas apply lets you apply a function to each row in a pandas DataFrame. Let's create a function that'll look at all the columns, find the max value, find the min value, and compute the difference for each row in the pandas DataFrame.

Start by creating a pandas DataFrame with three columns and three rows of data:

```
df = pd.DataFrame({"a": [23, 2, 8], "b": [99, 6, 1], "c": [1, 2,
3]})
```

Here are the contents of the DataFrame:

```
>>> df

   a   b   c
0  23  99  1
1  2   6   2
2  8   1   3
```

Define a `minmax` function that will take the difference between the max value and the min value and then use the pandas `apply()` method to run this function on each row in the DataFrame:

```
def minmax(x):
    return x.max() - x.min()
```

Here's the result:

```
>>> df.apply(minmax, axis=1)
0  98
1  4
```

```
2  7
dtype: int64
```

The pandas apply() function returns the result as a Series object.

```
>> type(df.apply(minmax, axis=1))
pandas.core.series.Series
```

Let's look at how to parallelize this function with Dask.

Parallelizing pandas apply with map_partitions

Convert the pandas DataFrame to a Dask DataFrame with two partitions:

```
ddf = dd.from_pandas(df, npartitions=2)
```

Create a minmax2 function that wraps the original minmax function and use map_partitions() to run it on each partition in a Dask DataFrame.

```
def minmax2(df):
    return df.apply(minmax, axis=1)
```

Map the function across all partitions in the Dask DataFrame:

```
>>> ddf.map_partitions(minmax2, meta=(None, "int64")).compute()
0  98
1  4
2  7
dtype: int64
```

map_partitions() runs the pandas apply() operation on all the partitions in parallel, so map_partitions() is a great way to parallelize a pandas apply() operation and make it run faster.

> **NOTE**
>
> [CAUTION] In the toy examples, the Dask version may not run faster than the pandas version because the data is so small and the Dask scheduler incurs minimal overhead. Remember that Dask is meant for large datasets where the benefits of parallel computing (processing speed) outweigh the costs (overhead of)

Calculating memory usage of a DataFrame with map_partitions

Let's look at two approaches for calculating the memory for each partition of a 662 million row dataset. You don't need to actually use this approach to compute the memory usage of a DataFrame because Dask has a built-in memory_usage() method that's more convenient. Nonetheless, this example is a good way to demonstrate the power of the map_partitions() method:

```
>>> ddf = dd.read_parquet(
        "s3://coiled-datasets/timeseries/20-years/parquet",
        storage_options={"anon": True, 'use_ssl': True}
    )
>>> ddf.map_partitions(lambda x:
x.memory_usage(deep=True).sum()).compute()
0       57061027
1       57060857
2       57059768
3       57059342
4       57060737
          ...
1090    57059834
1091    57061111
1092    57061001
1093    57058404
1094    57061989
Length: 1095, dtype: int64
```

This computation takes 124 seconds on a 5-node cluster.

Dask has a `sizeof()` function that estimates the size of each partition and runs faster.

```
>>> ddf.map_partitions(lambda x: dask.sizeof.sizeof(x)).compute()
0       56822960
1       57125360
2       56822960
3       57246320
4       57306800
          ...
1090    56974160
1091    57004400
1092    57337040
1093    56822960
```

```
1094    57004400
Length: 1095, dtype: int64
```

This takes 92 seconds to run, which is 21% faster than `memory_usage()` on the same dataset.

The `sizeof()` results are an approximation, but they're pretty close as you can see.

groupby aggregations

This section explains how to perform groupby() aggregations with Dask DataFrames.

You'll learn how to perform groupby() operations with one and many columns. You'll also learn how to compute aggregations like sum, mean, and count.

After you learn the basic syntax, we'll discuss the best practices when performing groupby() operations.

Dask DataFrame groupby sum

Let's read a sample dataset from S3 into a Dask DataFrame to perform some sample groupby computations. We will use Coiled to launch a Dask computation cluster with 5 nodes.

```
import coiled
import dask
import dask.dataframe as dd
cluster = coiled.Cluster(name="demo-cluster", n_workers=5)
client = dask.distributed.Client(cluster)
ddf = dd.read_parquet(
    "s3://coiled-datasets/h2o/G1_1e7_1e2_0_0/parquet",
    storage_options={"anon": True, "use_ssl": True},
    engine="pyarrow",
)
```

Let's groupby the values in the `id` column and then sum the values in the x column.

```
>>> ddf.groupby("id").x.sum().compute()
id
858     -8.741694
862      4.377649
863     -0.858438
866     -0.332073
869     -27.662715
        ...
```

You can also use an alternative syntax and get the same result.

```
ddf.groupby("id").agg({"x": "sum"}).compute()
```

agg takes a more complex code path in Dask, so you should generally stick with the simple syntax unless you need to perform multiple aggregations.

Dask DataFrame groupby for a single column is pretty straightforward. Let's look at how to groupby with multiple columns.

Dask DataFrame groupby multiple columns

Here's how to group by the `id` and `name` columns and then sum the values in x:

```
>>> ddf.groupby(["id", "name"]).x.sum().compute()
id      name
858     Xavier    -0.459693
862     Frank      0.409465
        Ingrid     1.067823
863     Bob        0.048593
866     Norbert   -0.051115
                  ...
```

You can pass a list to the Dask groupby method to group by multiple columns.

Now let's look at how to perform multiple aggregations after grouping.

Dask groupby multiple aggregations

Here's how to group by `id` and compute the sum of x and the mean of `y`:

```
>>> ddf.groupby("id").agg({"x": "sum", "y": "mean"}).compute()
```

id	x	y
815	-0.677002	-0.958081
816	0.850561	0.052172
819	-0.948970	-0.081699
821	0.131839	-0.689080
824	0.635101	0.326280
...	...	...

Figure 2-11. Figure title needed.

You can pass a dictionary to the `agg` method to perform different types of aggregations.

Let's turn our attention to how Dask implements groupby computations. Specifically, let's look at how Dask changes the number of partitions in the DataFrame when a groupby operation is performed. This is important because you need to manually set the number of partitions properly when the aggregated DataFrame is large.

How Dask groupby impacts npartitions

Dask doesn't know the contents of your DataFrame ahead of time. So it can't know how many groups the groupby operation will produce. By default, it assumes you'll have relatively few groups, so the number of rows is reduced so significantly that the result will fit comfortably in a single partition.

However, when your data has many groups, you'll need to tell Dask to split the results into multiple partitions in order to not overwhelm one unlucky worker.

Dask DataFrame groupby will return a DataFrame with a single partition by default. Let's look at a DataFrame, confirm it has multiple partitions, run a groupby operation, and then observe how the resulting DataFrame only has a single partition.

The DataFrame that we've been querying has 1,095 partitions:

```
>> ddf.npartitions
1095
```

Now let's run a groupby operation on the DataFrame and see how many partitions are in the result.

```
>>> res = ddf.groupby("id").x.sum()
>>> res.npartitions
1
```

Dask will output groupby results to a single partition Dask DataFrame by default. A single partition DataFrame is all that's needed in most cases. `groupby` operations usually reduce the number of rows in a DataFrame significantly so they can be held in a single partition DataFrame.

You can set the `split_out` argument to return a DataFrame with multiple partitions if the result of the groupby operation is too large for a single partition Dask DataFrame:

```
>>> res2 = ddf.groupby("id").x.sum(split_out=2)
>>> res2.npartitions
2
```

In this example, `split_out` was set to two, so the groupby operation results in a DataFrame with two partitions. **The onus is on you to properly set the `split_out` size when the resulting DataFrame is large.**

Performance considerations

Dask DataFrames are divided into many partitions, each of which is a pandas DataFrame. Dask performs groupby operations by running groupby on each of the individual pandas DataFrames and then aggregating all the results. The Dask DataFrame parallel execution of groupby on multiple subsets of the data makes it more scalable than pandas and often quicker too.

Memory usage

This section shows you how to compute the memory usage of a Dask DataFrame and how to develop a partitioning strategy based on the distribution of your data.

Dask DataFrames distribute data in partitions, so computations can be run in parallel. Each partition in a Dask DataFrame is a pandas DataFrame. This section explains how to measure the amount of data in each Dask partition. Intelligently distributing the data across partitions is important for performance.

There aren't hard-and-fast rules on optimal partition sizes. It depends on the computing power of the nodes in your cluster and the analysis you're running.

A general rule of thumb is to target 100 MB of data per memory partition in a cluster. This section shows you how to measure the distribution of data in your cluster so you know when and if you need to repartition.

Here's what you'll learn in this section:

1. Calculation memory usage of a small Dask DataFrame
2. Memory usage of a large Dask DataFrame

3. Filtering can cause partition imbalances
4. Assessing when a Dask DataFrame's memory usage is unevenly distributed
5. Fixing imbalances with repartitioning
6. Other ways to compute memory usage by partition

Memory usage of small Dask DataFrames

Create a small Dask DataFrame with two partitions:

```
import pandas as pd
from dask import dataframe as dd
df = pd.DataFrame({
    "nums": [1, 2, 3, 4, 5, 6],
    "letters": ["a", "b", "c", "d", "e", "f"]
})
ddf = dd.from_pandas(df, npartitions=2)
```

Print the data in each of the partitions:

```
>>> for i in range(ddf.npartitions):
        print(ddf.partitions[i].compute())
  nums letters
0     1       a
1     2       b
2     3       c
   nums letters
3     4       d
4     5       e
5     6       f
```

Use the pandas `memory_usage` method to print the bytes of memory used in each column of the first partition:

```
>>> ddf.partitions[0].memory_usage(deep=True).compute()
Index      128
letters    174
nums        24
dtype: int64
```

Print the total memory used by each partition in the DataFrame with the Dask `memory_usage_per_partition` method:

```
>>> ddf.memory_usage_per_partition(deep=True).compute()
0    326
1    330
dtype: int64
```

Both of these partitions are tiny because the entire DataFrame only contains six rows of data.

If deep is set to False then the memory usage of the object columns is not counted:

```
>>> ddf.memory_usage_per_partition(deep=False).compute()
0    176
1    180
dtype: int64
```

Calculating the memory usage of object columns is slow, so you can set `deep` to `False` and make the computation run faster. We care about how much memory all the columns are using, so our examples use `deep=True`.

Memory usage of large Dask DataFrame

Let's calculate the memory for each partition of the dataset.

```
>>> ddf = dd.read_parquet(
        "s3://coiled-datasets/timeseries/20-years/parquet",
        storage_options={"anon": True, 'use_ssl': True}
    )
>>> ddf.memory_usage_per_partition(deep=True).compute()
0       57061027
1       57060857
2       57059768
3       57059342
4       57060737
          ...
1090    57059834
1091    57061111
1092    57061001
1093    57058404
```

```
1094    57061989
Length: 1095, dtype: int64
```

The DataFrame has 1,095 partitions and each partition has 57 MB of data.

The data is evenly balanced across each partition in the DataFrame. There aren't lots of tiny, empty, or huge partitions. You probably don't need to repartition this DataFrame because all the memory partitions are reasonably sized and the data is evenly distributed.

Tips on managing memory

Let's see how Azeem can reduce the memory usage of the large Parquet DataFrame, so his analysis doesn't consume as many resources and runs more efficiently.

Azeem can limit the amount of memory his analysis takes by sending less data to the cluster or by using data types that are more memory efficient.

Let's take a look at how much memory Azeem's Parquet dataset uses in memory:

```
>>> from dask.utils import format_bytes
>>> format_bytes(ddf.memory_usage(deep=True).sum().compute())
'58.19 GiB'
```

The dataset takes 58 GB in memory. Let's see how much memory it takes, by column:

```
>>> ddf.memory_usage(deep=True).compute().apply(format_bytes)
Index    4.93 GiB
id       4.93 GiB
name    38.45 GiB
x        4.93 GiB
y        4.93 GiB
dtype: object
```

The name column is by far the most memory-greedy, requiring more memory than all the other columns combined. Let's take a look at the data types for this DataFrame to see why `name` is taking so much memory:

```
>>> ddf.dtypes
id      int64
name   object
x     float64
y     float64
dtype: object
```

`name` is an object column, which isn't surprising because object columns are notoriously memory hungry. Let's change the `name` column to a different type and see if that helps.

String types

Let's change the name column to be a string and see how that impacts memory usage of the DataFrame:

```
>>> ddf.name = ddf.name.astype("string[pyarrow]")
>>> ddf.memory_usage(deep=True).compute().apply(format_bytes)
Index    4.93 GiB
id       4.93 GiB
name     5.76 GiB
x        4.93 GiB
y        4.93 GiB
dtype: object
```

The name column only takes 5.76 GiB in memory now. It used to take 38.45 GiB. That's a significant 6.7x memory reduction. Avoid object columns whenever possible as they are very memory inefficient.

Smaller numeric types

You can also use smaller numeric types for columns that don't require 64 bits. Let's look at the values in the id column and see if it really needs to be typed as an int64.

Here's how to compute the min and max values in the id column:

```
>>> dd.compute(ddf.id.min(), ddf.id.max())
815, 119
```

We don’t need int64s for holding such small values. Let’s switch to int16 and quantify memory savings:

```
>>> ddf.id = ddf.id.astype("int16")
>>> ddf.memory_usage(deep=True).compute().apply(format_bytes)
Index    4.93 GiB
id       1.23 GiB
name    38.45 GiB
x        4.93 GiB
y        4.93 GiB
dtype: object
```

The id column used to take 4.93 GiB and now only takes 1.23 GiB. That’s 4 times smaller, which isn’t surprising because 16 bit numbers are four times smaller than 64 bit numbers.

We’ve seen how column types can reduce the memory requirements of a DataFrame. Now let’s look at how loading less data to the cluster also can reduce memory requirements.

Column pruning

Suppose you need to run a query that only requires the x column and won’t use the data in the other columns.

You don’t need to read all the data into the cluster, you can just read the x column with column pruning:

```
>>> ddf = dd.read_parquet(
        "s3://coiled-datasets/timeseries/20-years/parquet",
        storage_options={"anon": True, "use_ssl": True},
        engine="pyarrow",
        columns=["x"],
    )
>>> format_bytes(ddf.memory_usage(deep=True).sum().compute())
9.87 GiB
```

This DataFrame only contains the x column and the index:

```
>>> ddf.memory_usage(deep=True).compute().apply(format_bytes)
Index    4.93 GiB
```

```
x         4.93 GiB
dtype: object
```

Only storing a fraction of the columns obviously makes the overall memory footprint much lower.

Predicate pushdown filters

Parquet files also let you skip entire row groups for some queries which also limits the amount of data that's sent to the computation cluster.

Here's how to read the data and only include row groups that contain at least one value with an id greater than 1170:

```
ddf = dd.read_parquet(
    "s3://coiled-datasets/timeseries/20-years/parquet",
    storage_options={"anon": True, "use_ssl": True},
    engine="pyarrow",
    filters=[[('id', '>', 1170)]],
)
```

There isn't much data with an id greater than 1170 for this dataset, so this predicate pushdown filter greatly reduces the size of the data in memory:

```
>>> format_bytes(ddf.memory_usage(deep=True).sum().compute())
'5.99 kiB'
```

The more row groups that are excluded by the predicate pushdown filters, the smaller the DataFrame in memory.

Converting to number columns with to_numeric

This section explains how to convert Dask DataFrame object columns to floating point columns with `to_numeric()` and why it's more flexible than `astype()` in certain situations. This design pattern is especially useful when you're working with data in text based file formats like CSV. You'll often have to read in numeric columns stored in CSV files as object columns because of messy data and then convert the numeric columns to floating point values to null out the bad data. You can't perform numerical operations on columns that are typed as objects. You can use the tactics

outlined in this section for data munging in an extract, transform, & load (ETL) pipeline.

Cleaning data is often the first step of a data project. Luckily Dask has great helper methods like `to_numeric` that make it easy to clean the data and properly type the columns in your DataFrames.

Converting object column with to_numeric

Let's look at a simple example with a DataFrame that contains an invalid string value in a column that should only contain numbers. Let's start by creating a DataFrame with `nums` and `letters` columns:

```
import dask.dataframe as dd
import pandas as pd
df = pd.DataFrame({
"nums": [1, 2.8, 3, 4, "hi", 6],
"letters": ["a", "b", "c", "d", "e", "f"]
})
ddf = dd.from_pandas(df, npartitions=2)
```

Now let's print the contents of the DataFrame so it's easy to visualize:

```
>>> print(ddf.compute())
 nums letters
0    1       a
1  2.8       b
2    3       c
3    4       d
4   hi       e
5    6       f
```

Notice that row 4 in the nums column has the value "hi". That's a string value that Python cannot convert into a numerical value.

Let's look at the data types of the columns and see that Dask is treating both `nums` and `letters` as object type columns:

```
>>> ddf.dtypes
nums       object
letters    object
```

Let's convert the `nums` column to be a number column with `to_numeric`:

```
>>> ddf["nums"] = dd.to_numeric(ddf["nums"], errors="coerce")
>>> ddf.dtypes
nums        int64
letters    object
>>> print(ddf.compute())
  nums letters
0   1.0       a
1   2.8       b
2   3.0       c
3   4.0       d
4   NaN       e
5   6.0       f
```

Dask has conveniently nulled out the "hi" value in row 4 to be NaN. Nulling out values that cannot be easily converted to numerical values is often what you'll want. Alternatively, you can set `errors="raise"` to raise an error when a value can't be cast to numeric dtype.

Limitations of astype

Many beginning Dask users tend to use the `astype` method to convert object columns to numeric columns. This has important limitations.

Let's create another DataFrame to see when `astype` can be used to convert from object columns to numeric columns and when it falls short:

```
>>> df2 = pd.DataFrame({
        "n1": ["bye", 2.8, 3],
        "n2": ["7.7", "8", 9.2]
})
>>> ddf2 = dd.from_pandas(df, npartitions=2)
>>> print(ddf2.compute())
   n1    n2
0  bye  7.7
1  2.8    8
2    3  9.2
```

You can run `ddf2.dtypes` to see that both `n1` and `n2` are object columns:

```
>>> ddf2.dtypes
n1    object
n2    object
```

`n2` is an object type column because it contains string and float values.

Let's convert `n2` to be a `float64` column using `astype`:

```
>>> ddf2["n2"] = ddf2["n2"].astype("float64")
>>> ddf2.dtypes
n1     object
n2    float64
dtype: object
>>> print(ddf2.compute())
   n1   n2
0  bye  7.7
1  2.8  8.0
2    3  9.2
```

`astype` can convert `n2` to be a float column without issue.

Now let's try to convert `n1` to be a float column with `astype`:

```
>>> ddf2["n1"] = ddf2["n1"].astype("float64")
>>> print(ddf2.compute())
```

This errors out with the following error message:

```
ValueError: could not convert string to float: 'bye'
```

`astype` raises errors when columns contain string values that cannot be converted to numbers. It doesn't coerce string values to `NaN`.

`to_numeric` also has the same default behavior and this code will error out as well:

```
>>> ddf2["n1"] = dd.to_numeric(ddf["n1"])
>>> print(ddf2.compute())
```

```
"ValueError: Unable to parse string "bye" at position 0".
```

You need to set errors="coerce" to successfully invoke `to_numeric`:

```
>>> ddf2["n1"] = dd.to_numeric(ddf["n1"], errors="coerce")
>>> print(ddf2.compute())
   n1   n2
0  NaN  7.7
1  2.8  8.0
2  3.0  9.2
```

Best practices for numeric columns

Dask makes it easy to convert object columns into number columns with `to_numeric`.

`to_numeric` is customizable with different error behavior when values cannot be converted to numbers. You can coerce these values to `NaN`, raise an error, or ignore these values. Choose the behavior that works best for your application.

It's good practice to make sure all your numeric columns are properly typed before performing your analysis, so you don't get weird downstream bugs.

Vertically union Dask DataFrames

This section teaches you how to union Dask DataFrames vertically with `concat` and the important related technical details. Vertical concatenation combines DataFrames like the SQL UNION operator combines tables which is common when joining datasets for reporting and machine learning. It's useful whenever you have two tables with identical schemas that you'd like to combine into a single DataFrame.

The tactics outlined in this section will help you combine two DataFrame with the same, or similar, schemas into a single DataFrame. It's a useful design pattern to have in your toolkit.

Here's how this section on vertical concatenations is organized:

- Concatenating DataFrames with identical schemas / dtypes
- Interleaving partitions to maintain divisions integrity
- Concatenating DataFrames with different schemas

- Concatenating large DataFrames

Concatenate DataFrames with identical schemas

Create two Dask DataFrames with identical schemas:

```
import dask.dataframe as dd
import pandas as pd
df = pd.DataFrame({
    "nums": [1, 2, 3, 4, 5, 6],
    "letters": ["a", "b", "c", "d", "e", "f"]
})
ddf1 = dd.from_pandas(df, npartitions=2)
df = pd.DataFrame({"nums": [88, 99], "letters": ["xx", "yy"]})
ddf2 = dd.from_pandas(df, npartitions=1)
```

Now concatenate both the DataFrames into a single DataFrame:

```
ddf3 = dd.concat([ddf1, ddf2])
```

Print the contents of `ddf3` to verify it contains all the rows from `ddf1` and `ddf2`:

```
>>> print(ddf3.compute())
  nums letters
0     1       a
1     2       b
2     3       c
3     4       d
4     5       e
5     6       f
0    88      xx
1    99      yy
```

`ddf1` has two partitions and `ddf2` has one partition. `ddf1` and `ddf2` are combined to `ddf3`, which has three total partitions:

```
>>> ddf3.npartitions
3
```

Dask can use information on divisions to speed up certain queries. The creation of `ddf3` above wiped out information about DataFrame divisions.

Let's see how we can interleave partitions when concatenating DataFrames to avoid losing divisions data.

Interleaving partitions

Let's revisit our example with a focus on DataFrame divisions to illustrate how `concat` wipes out the DataFrame divisions by default.

Recreate the `ddf1` DataFrame and look at its divisions:

```
>>> df = pd.DataFrame({
        "nums": [1, 2, 3, 4, 5, 6],
        "letters": ["a", "b", "c", "d", "e", "f"]
})
ddf1 = dd.from_pandas(df, npartitions=2)
>>> ddf1.divisions
(0, 3, 5)
```

Here's how to interpret this divisions output:

- The first partition has index values between 0 and 2
- The second partitions has index values between 3 and 5

Let's print every partition of the DataFrame to visualize the actual data and reason about the division's values:

```
>>> def print_partitions(ddf):
        for i in range(ddf.npartitions):
            print(ddf.partitions[i].compute())
>>> print_partitions(ddf1)
  nums letters
0     1       a
1     2       b
2     3       c
   nums letters
3     4       d
4     5       e
5     6       f
```

Let's recreate `ddf2` and view its divisions too:

```
>>> df = pd.DataFrame({"nums": [88, 99], "letters": ["xx", "yy"]})
```

```
>>> ddf2 = dd.from_pandas(df, npartitions=1)
>>> ddf2.divisions
(0, 1)
```

`ddf2` has a single partition with index values between zero and one:

```
>>> print_partitions(ddf2)
   nums letters
0    88      xx
1    99      yy
```

Let's concatenate the DataFrames and see what happens with the divisions:

```
>>> ddf3 = dd.concat([ddf1, ddf2])
>>> ddf3.divisions
(None, None, None, None)
```

Dask has lost all information about divisions for `ddf3` and won't be able to use divisions related optimizations for subsequent computations.

You can set `interleave_partitions` to `True` when concatenating DataFrames to avoid losing information about divisions:

```
>>> ddf3_interleave = dd.concat([ddf1, ddf2],
interleave_partitions=True)
>>> ddf3_interleave.divisions
(0, 1, 3, 5)
```

Take a look at how the data is distributed across partitions in `ddf3_interleave`:

```
>>> print_partitions(ddf3_interleave)
  nums letters
0     1       a
0    88      xx
   nums letters
1     2       b
2     3       c
1    99      yy
   nums letters
3     4       d
4     5       e
5     6       f
```

Dask can optimize certain computations when divisions exist. Set `interleave_partitions` to `True` if you'd like to take advantage of these optimizations after concatenating DataFrames.

Concatenating DataFrames with different schemas

You can also concatenate DataFrames with different schemas. Let's create two DataFrames with different schemas, concatenate them, and see how Dask behaves.

Start by creating the two DataFrames:

```
df = pd.DataFrame(
    {
        "animal": ["cat", "dolphin", "shark", "starfish"],
        "is_mammal": [True, True, False, False],
    }
)
ddf1 = dd.from_pandas(df, npartitions=2)
df = pd.DataFrame({"animal": ["hippo", "lion"], "likes_water":
[True, False]})
ddf2 = dd.from_pandas(df, npartitions=1)
```

Concatenate the DataFrames and print the result:

```
>>> ddf3 = dd.concat([ddf1, ddf2])
>>> print(ddf3.compute())
    animal is_mammal likes_water
0       cat      True         NaN
1   dolphin      True         NaN
2     shark     False         NaN
3  starfish     False         NaN
0     hippo       NaN        True
1      lion       NaN       False
```

Dask fills in the missing values with `NaN` to make the concatenation possible.

Concatenating large DataFrames

Lets create a Dask cluster and concatenate the 20-year DataFrame with a DataFrame that contains 311 million row of timeseries data from January 1,

1990 till December 31, 1999. This section demonstrates that `concat` can scale to multi-node workflows.

Create a 5-node Coiled cluster and read in a Parquet dataset into a DataFrame:

```
import coiled
import dask
cluster = coiled.Cluster(name="concat-cluster", n_workers=5)
client = dask.distributed.Client(cluster)
ddf1990s = dd.read_parquet(
    "s3://coiled-datasets/timeseries/7d/parquet/1990s",
    storage_options={"anon": True, "use_ssl": True},
    engine="pyarrow"
)
```

Run `ddf1990s.head()` to visually inspect the contents of the DataFrame:

```
>>> ddf1990s.head()
```

timestamp	id	name	x	y
2000-01-01 00:00:00	1008	Dan	-0.259374	-0.118314
2000-01-01 00:00:01	987	Patricia	0.069601	0.755351
2000-01-01 00:00:02	980	Zelda	-0.281843	-0.510507
2000-01-01 00:00:03	1020	Ursula	-0.569904	0.523132
2000-01-01 00:00:04	967	Michael	-0.251460	0.810930

Figure 2-12. Figure title needed.

Let's run some analytical queries on `ddf1990s` to better understand the data it contains:

```
>>> len(ddf1990s)
311,449,600
>>> ddf1990s.npartitions
552
```

Now let's look at our original Parquet dataset with data from January 1, 2000 till December 31, 2020:

```
>>> len(ddf)
661,449,600
>>> ddf.npartitions
1050
```

Concatenate the two DataFrames and inspect the contents of the resulting DataFrame:

```
>>> ddf = dd.concat([ddf1990s, ddf])
>>> len(ddf)
972,899,200

>>> ddf.npartitions
1602
>>> ddf.divisions
(Timestamp('1990-01-01 00:00:00'),
 Timestamp('1990-01-08 00:00:00'),
 Timestamp('1990-01-15 00:00:00'),
 …
 Timestamp('2020-12-17 00:00:00'),
 Timestamp('2020-12-24 00:00:00'),
 Timestamp('2020-12-30 23:59:59'))
```

These DataFrames were concatenated without `interleave_partitions=True` and the divisions metadata was not lost like we saw earlier.

The DataFrames in this example don't have any overlapping divisions, so you don't need to set `interleave_partitions=True`.

Writing Data with Dask DataFrames

You've completed all your analyses and want to store the results sitting in your Dask DataFrames.You can write the contents of Dask DataFrames to CSV files, Parquet files, HDF files, SQL tables and convert them into other Dask collections like Dask Bags, Dask Arrays and Dask Delayed. In this chapter we will cover the first 2 options (CSV and Parquet). See the Dask documentation for more information on the other options.

> **NOTE**
>
> [NOTE] Depending on the analysis you are running, your results may at this point be small enough to fit into local memory. For example if you've run groupby aggregations on your dataset. In this case, you should stop using Dask and switch back to normal pandas by calling `results.compute()`.

There are a lot of different keyword arguments to the `to_csv` and `to_parquet` functions, more than we're going to cover here. However there are a few essential kwargs you should know about and we're going to cover them below: options for file compression, writing to multiple vs single files, and partitioning on specific columns.

Let's start with the basics.

You can write a Dask DataFrame out to a CSV or Parquet file using the `to_csv` and `to_parquet` method, respectively:

```
ddf.to_parquet("filename.parquet")
ddf.to_csv("filename.csv")
```

Dask DataFrames can be written to a variety of different sources and file formats. We'll talk about working with different file formats in more detail in Chapter 7.

File Compression

Both the to_csv and to_parquet writers have multiple options for file compression. The to_csv writer allows the following compression options: gzip, bz2 and xz.

The to_parquer writer defaults to 'snappy'. It also accepts other Parquet compression options like gzip, and blosc. You can also pass a dictionary to this keyword argument to map columns to compressors, for example: `{"name": "gzip", "values": "snappy"}`. We recommend using the default "snappy" compressor.

to_csv: single_file

Dask DataFrames consist of multiple partitions. By default, writing a Dask DataFrame to CSV will write each partition to a separate CSV file.

Let's illustrate. We'll start by creating a partitioned Dask DataFrame:

```
>>> df = pd.DataFrame({
            "nums": [1, 2, 3, 4, 5, 6],
            "letters": ["a", "b", "c", "d", "e", "f"]
    })
>>> ddf = dd.from_pandas(df, npartitions=2)
>>> ddf.npartitions
2
```

Let's inspect the contents of the first partition:

```
>>> ddf.partitions[0].compute()
     nums   letters
0    1      a
1    2      b
2    3      c
```

Now let's write the whole Dask DataFrame out to CSV with default settings:

```
ddf.to_csv("data.csv")
```

Now switch to a terminal, `cd` into the data.csv folder and `ls` the contents:

```
>>> $ cd data.csv
>>> $ ls
0.part  1.part
```

We clearly see two partial CSV files here. To confirm, let's load in just the 0.part file and inspect the contents:

```
>>> ddf_1 = dd.read_csv('data.csv/0.part')
>>> ddf_1.compute()
     0: unnamed   nums   letters
0    0            1      a
```

```
1    1            2      b
2    2            3      c
```

As expected, the content matches that of the first partition of our original Dask DataFrame ddf **except** for the fact that Dask seems to have duplicated the Index column. This happens because, just like in pandas, to_csv writes the index as a separate column by default. You can change this setting by setting the index keyword to False, just like you would in pandas:

```
>>> ddf.to_csv("data.csv", index=False)
>>> ddf_1 = dd.read_csv('data.csv/0.part')
>>> ddf_1.compute()
     nums   letters
0    1      a
1    2      b
2    3      c
```

NOTE

[NOTE] The dask.dataframe.to_csv writer, just like the dask.dataframe.read_csv reader, accepts many of the same keyword arguments as their pandas equivalents.

Fortunately, Dask makes it easy for you to read multiple CSV files located in a single directory into a Dask DataFrame, using the * character as a glob string:

```
>>> ddf = dd.read_csv('data.csv/*')
>>> ddf.compute()
     nums   letters
0    1      a
1    2      b
2    3      c
0    4      d
1    5      e
2    6      f
```

Having your data scattered over multiple CSV files may seem impractical at first, especially if you're used to working with CSV files in pandas. But remember that you're likely using Dask because either your dataset is too

large to fit into memory or you want to enjoy the benefits of parallelism – or both! Having the CSV file split up into smaller parts means Dask can process the file in parallel and maximize performance.

If, however, you want to write your data out to a single CSV file, you can change the default setting by setting `single_file` to `True`:

```
ddf.to_csv(
"single_file.csv",
index=False,
single_file=True
)
```

> **NOTE**
>
> [CAUTION] The to_csv writer clobbers existing files in the event of a name conflict. Be careful whenever you're writing to a folder with existing data, especially if you're using the default file names.

to_parquet: engine

The engine keyword can be used to choose which Parquet library to use for writing Dask DataFrames to Parquet. We strongly recommend using the pyarrow engine, which will be the default starting from

to_parquet: partition_on

Dask DataFrame's to_parquet writer allows you to partition the resulting Parquet files according to the values of a particular column.

Let's illustrate with an example. We'll Create a Dask DataFrame with letter and number columns and write it out to disk, partitioning on the letter column:

```
df = pd.DataFrame({
"letter": ["a", "b", "c", "a", "a", "d"],
"number": [1, 2, 3, 4, 5, 6]
})
```

```
ddf = dd.from_pandas(df, npartitions=3)
ddf.to_parquet(
"output/partition_on",
partition_on="letter"
)
```

Here are the files that are written to disk:

```
output/partition_on
  letter=a/
    part.0.parquet
    part.1.parquet
    part.2.parquet
  letter=b/
    part.0.parquet
  letter=c/
    part.1.parquet
  letter=d/
    part.2.parquet
```

Organizing the data in this directory structure lets you easily skip files for certain read operations. For example, if you only want the data where the letter equals a, then you can look in the tmp/partition/1/letter=a directory and skip the other Parquet files.

The letter column is referred to as the partition key in this example.

Here's how to read the data in the letter=a disk partition:

```
>>> ddf = dd.read_parquet(
       "tmp/partition/1",
       engine="pyarrow",
       filters=[("letter", "==", "a")]
    )
>>> print(ddf.compute())
  number letter
0       1      a
3       4      a
4       5      a
```

Dask is smart enough to apply the partition filtering based on the filters argument. Dask only reads the data from output/partition_on/letters=a into the DataFrame and skips all the files in other partitions.

> **NOTE**
>
> [CAUTION] Disk partitioning improves performance for queries that filter on the partition key. It usually hurts query performance for queries that don't filter on the partition key.

Example query that runs faster on disk partitioned lake:

```
ddf = dd.read_parquet(
   "output/partition_on",
   engine="pyarrow",
   filters=[("letter", "==", "a")]
)
```

Example query that runs slower on disk partitioned lake:

```
ddf = dd.read_parquet(
    "output/partition_on",
    engine="pyarrow"
)
ddf.loc[ddf["number"] == 2]
```

The performance drag from querying a partitioned lake without filtering on the partition key depends on the underlying filesystem. Unix-like filesystems are good at performing file listing operations on nested directories. So you might not notice much of a performance drag on your local machine.

Cloud-based object stores, like AWS S3, are not Unix-like filesystems. They store data as key value pairs and are slow when listing nested files with a wildcard character (a.k.a. globbing).

You need to carefully consider your organization's query patterns when evaluating the costs/benefits of disk partitioning. If you are always filtering on the partition key, then a partitioned lake is typically best. Disk partitioning might not be worth it if you only filter on the partitioned key sometimes.

Other Keywords

You are now familiar with the most important keyword arguments to the to_csv and to_parquer writers. These tools will help you write your Dask DataFrames to the two most popular tabular file formats effectively and with maximum performance.

See the Dask documentation for more information on the other keyword arguments. See Chapter 7 for more details on working with other file formats.

Summary

This chapter concludes the Dask DataFrame section of this book. To recap, in Chapter 2 we worked through a real-world end-to-end example to illustrate how Dask DataFrames help you process massive amounts of tabular data to gain valuable insights. Chapter 3 took a step back to explain the architecture of Dask DataFrame and how it transcends the scalability limitations of pandas. Finally, this chapter has given you in-depth tools and a collection of best practices for processing tabular data with Dask. You could think of this final chapter as the definitive guide for working with Dask DataFrames, presented to you by the team that has spent years building, maintaining and optimizing Dask. Taken together, these three chapters contain everything you need to take the Dask DataFrame car off the beaten track and forge your own trails forward.

In this chapter, you learned:

- How to read data into Dask DataFrames from various sources and file formats
- How to execute common data-processing operations with Dask DataFrames, including groupby aggregations, joins, converting data types, and mapping custom Python functions over your data.
- How to optimize your data structure for maximum performance by setting the index, repartitioning, and managing memory usage.

- How to write your processed data out to CSV and Parquet.

The next two chapters will cover working with array data using Dask Array. Chapter 5 will work through a real-world end-to-end example, and Chapter 6 will combine an explanation of the Dask Array architecture with our definitive collection of best practices.