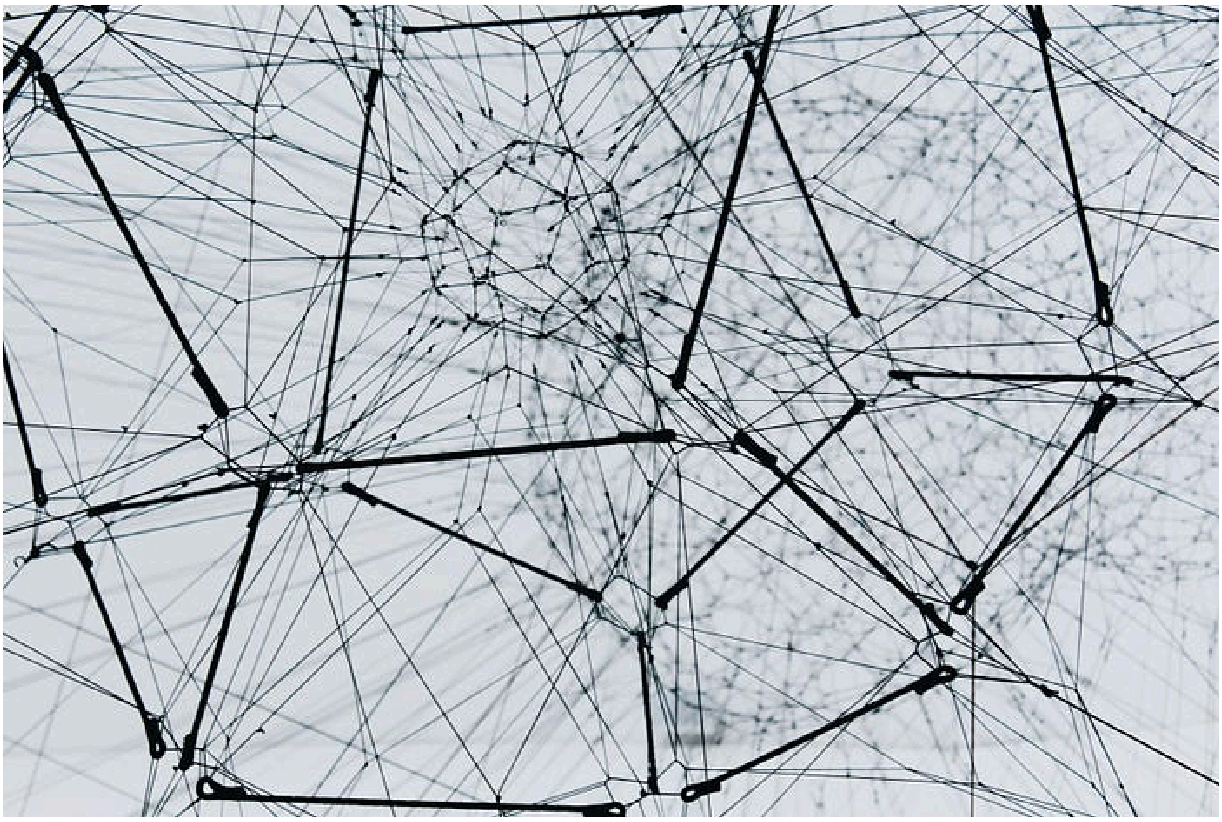
Freedium

< Go to the original



Python One Billion Row Challenge—From 10 Minutes to 4 Seconds

The one billion row challenge is exploding in popularity. How well does Python stack up?

Dario Radečić

Follow

Towards Data Science · ~10 min read · May 8, 2024 (Updated: May 8, 2024) ·
Free: No

The question of how fast a programming language can go through and aggregate 1 billion rows of data has been gaining traction lately. **Python**, not being the most performant language out there, naturally **doesn't stand a chance** — especially since the currently top-performing Java implementation takes only 1.535 seconds!

then see what happens if you use external libraries and better-suited file formats.

I've run all the scripts 5 times and averaged the results.

As for the hardware, I'm using a *16" M3 Pro Macbook Pro with 12 CPU cores and 36 GB of RAM*. Your results may vary if you decide to run the code, but hopefully, you should see similar percentage differences between implementations.

Code

- GitHub Repo — 1 Billion Row Challenge in Python

What is the 1 Billion Row Challenge?

The idea behind the 1 Billion Row Challenge (1BRC) is simple — go through a `.txt` file that contains arbitrary temperature measurements and calculate summary statistics for each station (min, mean, and max). The only issues are that you're working with 1 billion rows and that the data is stored in an uncompressed `.txt` format (13.8 GB).

The dataset is generated by the `data/createMeasurements.py` script on my GitHub repo. I've copied the script from the source, just to have everything in the same place.

Once you generate the dataset, you'll end up with a 13.8 GB semicolon-separated text file with two columns — **station name** and **measurement.**

Freedium

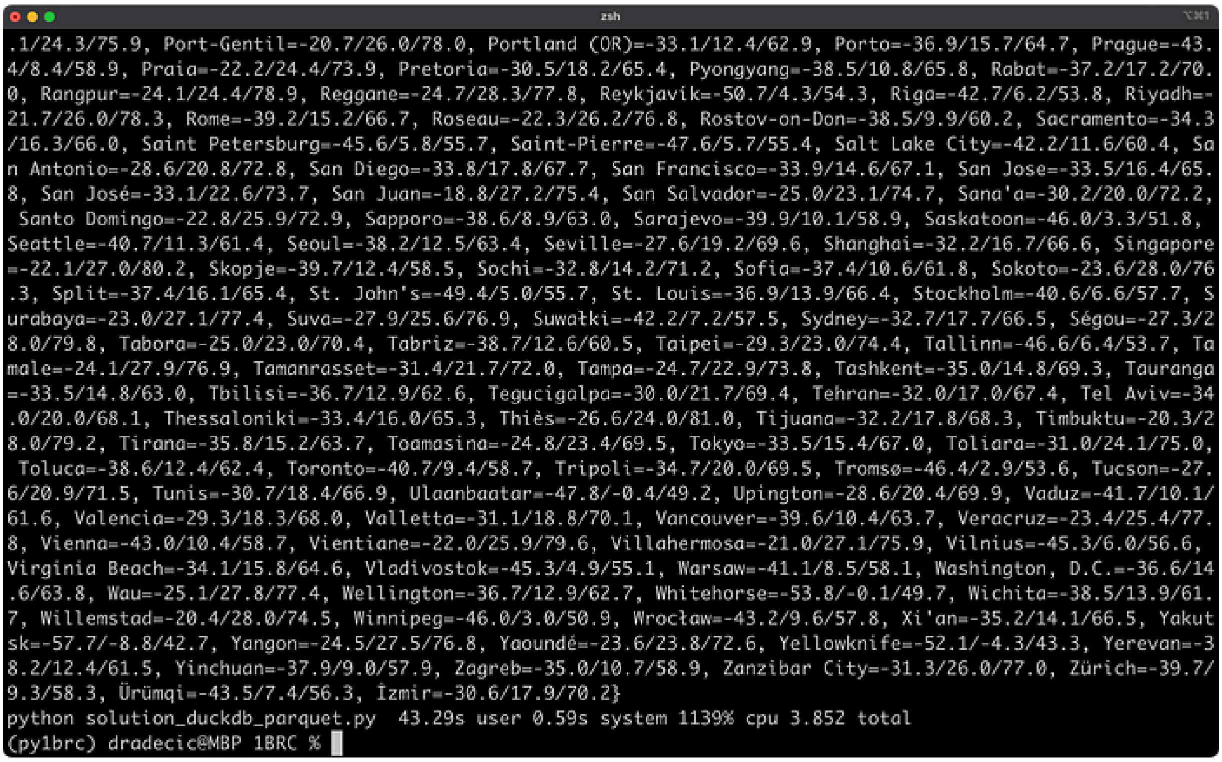


Image 1 — Sample results (image by author)

The actual output format is somewhat different from one implementation to the other, but this is the one I found proposed by the official Python repo.

You now know what the challenge is, so let's dive into the implementation next!

1 Billion Row Challenge — Pure Python Implementation

This is the only section in which I plan to obey the challenge rules. The reason is simple — Python doesn't stand a chance with its standard library, and everyone in the industry relies heavily on third-party packages.

Single-Core Implementation

Freedium

and max calculations, but mean requires keeping track of the count and then dividing the results.

Copy

```
# https://github.com/ifnesi/1brc#submitting
# Modified the multiprocessing version

def process_file(file_name: str):
    result = dict()

    with open(file_name, "rb") as f:
        for line in f:
            location, measurement = line.split(b";")
            measurement = float(measurement)
            if location not in result:
                result[location] = [
                    measurement,
                    measurement,
                    measurement,
                    1,
                ]
            else:
                _result = result[location]
                if measurement < _result[0]:
                    _result[0] = measurement
                if measurement > _result[1]:
                    _result[1] = measurement
                _result[2] += measurement
                _result[3] += 1

    print("{", end="")
    for location, measurements in sorted(result.items()):
        print(
            f"{location.decode('utf8')}={measurements[0]:.1f}/{
            end=", ",
        )
    print("\b\b} ")
```

Multi-core implementation

The same idea as before, but now you need to split the text file into equally sized chunks and process them in parallel. Here, you compute the statistics for each chunk and then combine the results.

Copy

```
# Code credits: https://github.com/ifnesi/1brc#submitting

import os
import multiprocessing as mp

def get_file_chunks(
    file_name: str,
    max_cpu: int = 8,
) -> list:
    """Split flie into chunks"""
    cpu_count = min(max_cpu, mp.cpu_count())

    file_size = os.path.getsize(file_name)
    chunk_size = file_size // cpu_count

    start_end = list()
    with open(file_name, "r+b") as f:

        def is_new_line(position):
            if position == 0:
                return True
            else:
                f.seek(position - 1)
                return f.read(1) == b"\n"

        def next_line(position):
            f.seek(position)
            f.readline()
            return f.tell()
```

```
            chunk_end = min(file_size, chunk_start + chunk_size)

            while not is_new_line(chunk_end):
                chunk_end -= 1

            if chunk_start == chunk_end:
                chunk_end = next_line(chunk_end)

            start_end.append(
                (
                    file_name,
                    chunk_start,
                    chunk_end,
                )
            )

            chunk_start = chunk_end

    return (
        cpu_count,
        start_end,
    )

def _process_file_chunk(
    file_name: str,
    chunk_start: int,
    chunk_end: int,
) -> dict:
    """Process each file chunk in a different process"""
    result = dict()
    with open(file_name, "rb") as f:
        f.seek(chunk_start)
        for line in f:
            chunk_start += len(line)
            if chunk_start > chunk_end:
                break
            location, measurement = line.split(b";")
            measurement = float(measurement)
            if location not in result:
                result[location] = [
                    measurement,
                    measurement,
                    measurement,
                    1,
```

```python
                _result = result[location]
                if measurement < _result[0]:
                    _result[0] = measurement
                if measurement > _result[1]:
                    _result[1] = measurement
                _result[2] += measurement
                _result[3] += 1
    return result

def process_file(
    cpu_count: int,
    start_end: list,
) -> dict:
    """Process data file"""
    with mp.Pool(cpu_count) as p:
        # Run chunks in parallel
        chunk_results = p.starmap(
            _process_file_chunk,
            start_end,
        )

    # Combine all results from all chunks
    result = dict()
    for chunk_result in chunk_results:
        for location, measurements in chunk_result.items():
            if location not in result:
                result[location] = measurements
            else:
                _result = result[location]
                if measurements[0] < _result[0]:
                    _result[0] = measurements[0]
                if measurements[1] > _result[1]:
                    _result[1] = measurements[1]
                _result[2] += measurements[2]
                _result[3] += measurements[3]

    # Print final results
    print("{", end="")
    for location, measurements in sorted(result.items()):
        print(
            f"{location.decode('utf8')}={measurements[0]:.1f}/{
            end=", ",
        )
    print("\b\b} ")
```

```
    cpu_count, *start_end = get_file_chunks("data/measurements.
    process_file(cpu_count=cpu_count, start_end=start_end[0])
```

PyPy implementation

Leverages the multiprocessing implementation to a large extent, but uses PyPy instead of CPython. This allows you to use a just-in-time compiler for your Python code. Great in all cases that don't depend on CPython extensions.

Copy

```
# Code credits: https://github.com/ifnesi/1brc#submitting

import os
import multiprocessing as mp


def get_file_chunks(
    file_name: str,
    max_cpu: int = 8,
) -> list:
    """Split flie into chunks"""
    cpu_count = min(max_cpu, mp.cpu_count())

    file_size = os.path.getsize(file_name)
    chunk_size = file_size // cpu_count

    start_end = list()
    with open(file_name, "r+b") as f:

        def is_new_line(position):
            if position == 0:
                return True
            else:
                f.seek(position - 1)
                return f.read(1) == b"\n"
```

```
            f.seek(position)
            f.readline()
            return f.tell()

        chunk_start = 0
        while chunk_start < file_size:
            chunk_end = min(file_size, chunk_start + chunk_size

            while not is_new_line(chunk_end):
                chunk_end -= 1

            if chunk_start == chunk_end:
                chunk_end = next_line(chunk_end)

            start_end.append(
                (
                    file_name,
                    chunk_start,
                    chunk_end,
                )
            )

            chunk_start = chunk_end

    return (
        cpu_count,
        start_end,
    )


def _process_file_chunk(
    file_name: str,
    chunk_start: int,
    chunk_end: int,
    blocksize: int = 1024 * 1024,
) -> dict:
    """Process each file chunk in a different process"""
    result = dict()

    with open(file_name, "r+b") as fh:
        fh.seek(chunk_start)

        tail = b""
        location = None
```

Freedium

```
while byte_count > 0:
    if blocksize > byte_count:
        blocksize = byte_count
    byte_count -= blocksize

    index = 0
    data = tail + fh.read(blocksize)
    while data:
        if location is None:
            try:
                semicolon = data.index(b";", index)
            except ValueError:
                tail = data[index:]
                break

            location = data[index:semicolon]
            index = semicolon + 1

        try:
            newline = data.index(b"\n", index)
        except ValueError:
            tail = data[index:]
            break

        value = float(data[index:newline])
        index = newline + 1

        if location not in result:
            result[location] = [
                value,
                value,
                value,
                1,
            ]  # min, max, sum, count
        else:
            _result = result[location]
            if value < _result[0]:
                _result[0] = value
            if value > _result[1]:
                _result[1] = value
            _result[2] += value
            _result[3] += 1

        location = None
```

Freedium

```python
def process_file(
    cpu_count: int,
    start_end: list,
) -> dict:
    """Process data file"""
    with mp.Pool(cpu_count) as p:
        # Run chunks in parallel
        chunk_results = p.starmap(
            _process_file_chunk,
            start_end,
        )

    # Combine all results from all chunks
    result = dict()
    for chunk_result in chunk_results:
        for location, measurements in chunk_result.items():
            if location not in result:
                result[location] = measurements
            else:
                _result = result[location]
                if measurements[0] < _result[0]:
                    _result[0] = measurements[0]
                if measurements[1] > _result[1]:
                    _result[1] = measurements[1]
                _result[2] += measurements[2]
                _result[3] += measurements[3]

    # Print final results
    print("{", end="")
    for location, measurements in sorted(result.items()):
        print(
            f"{location.decode('utf-8')}={measurements[0]:.1f}/
            end=", ",
        )
    print("\b\b} ")


if __name__ == "__main__":
    cpu_count, *start_end = get_file_chunks("data/measurements.
    process_file(cpu_count, start_end[0])
```

Freedium

1BRC Pure Python Results

As for the results, well, take a look for yourself:

None

Image 2 — Pure Python implementation results (image by author)

That's pretty much all you can squeeze out of Python's standard library. Even the PyPy implementation is over 11 times slower than the fastest Java implementation. So no, Python won't win any speed contest any time soon.

Speeding Things Up — Using 3rd Party Python Libraries

But what if you rely on third-party libraries? I said it before and I'll say it again — it's against the rules of the competition — but I'm beyond it at this point. I just want to make it run faster. No one's going to restrict me to Python's standard library on my day job anyway.

Pandas

A must-know data analysis library for any Python data professional. Not nearly the fastest one, but has a far superior ecosystem. I've used Pandas 2.2.2 with the PyArrow engine when reading the text file.

Copy

```
import pandas as pd

df = (
    pd.read_csv("data/measurements.txt", sep=";", header=None,
        .groupby("station_name")
```

```
df = df.reset_index()
df.columns = ["station_name", "min_measurement", "mean_measurem
df = df.sort_values("station_name")

print("{", end="")
for row in df.itertuples(index=False):
    print(
        f"{row.station_name}={row.min_measurement:.1f}/{row.mea
        end=", "
    )
print("\b\b} ")
```

Dask

Almost identical API to Pandas, but is lazy evaluated. You can use Dask to scale Pandas code across CPU cores locally or across machines on a cluster.

Copy

```
import dask.dataframe as dd

df = (
    dd.read_csv("data/measurements.txt", sep=";", header=None,
        .groupby("station_name")
        .agg(["min", "mean", "max"])
        .compute()
)

df.columns = df.columns.get_level_values(level=1)
df = df.reset_index()
df.columns = ["station_name", "min_measurement", "mean_measurem
df = df.sort_values("station_name")

print("{", end="")
for row in df.itertuples(index=False):
    print(
        f"{row.station_name}={row.min_measurement:.1f}/{row.mea
```

Polars

Similar to Pandas, but has a multi-threaded query engine written in Rust and offers order of operation optimization. I've written about it previously.

Copy

```
# Code credits: https://github.com/ifnesi/1brc#submitting

import polars as pl

df = (
    pl.scan_csv("data/measurements.txt", separator=";", has_hea
        .group_by("station_name")
        .agg(
            pl.min("measurement").alias("min_measurement"),
            pl.mean("measurement").alias("mean_measurement"),
            pl.max("measurement").alias("max_measurement")
        )
        .sort("station_name")
        .collect(streaming=True)
)

print("{", end="")
for row in df.iter_rows():
    print(
        f"{row[0]}={row[1]:.1f}/{row[2]:.1f}/{row[3]:.1f}",
        end=", "
    )
print("\b\b} ")
```

DuckDB

from the shell or over 10 different programming languages. In most of them, you can choose between a traditional analytical interface and a SQL interface. I've written about it previously.

Copy

```
import duckdb

with duckdb.connect() as conn:
    data = conn.sql("""
        select
            station_name,
            min(measurement) as min_measurement,
            cast(avg(measurement) as decimal(8, 1)) as mean_mea
            max(measurement) as max_measurement
        from read_csv(
            'data/measurements.txt',
            header=false,
            columns={'station_name': 'varchar', 'measurement':
            delim=';',
            parallel=true
        )
        group by station_name
        order by station_name
    """)

    print("{", end="")
    for row in sorted(data.fetchall()):
        print(
            f"{row[0]}={row[1]}/{row[2]}/{row[3]}",
            end=", ",
        )
    print("\b\b} ")
```

1BRC Third-Party Library Results

Freedium

None

Image 3 — Python data analysis libraries runtime results (image by author)

Pandas is slow — no surprises here. Dask offers pretty much the same performance as multi-core Python implementation, but with around 100 lines of code less. Polars and DuckDB reduce the runtime to below 10 seconds, which is impressive!

Going One Step Further — Ditching .txt for .parquet

There's still one bit of performance gain we can squeeze out, and that's changing the data file format. The `data/convertToParquet.py` file in the repo will do just that.

The idea is to go from uncompressed and unoptimized 13.8 GB of text data to a compressed and columnar-oriented 2.51 GB Parquet file.

The libraries remain the same, so it doesn't make sense to explain them again. I'll just provide the source code:

Pandas

Copy

```
import pandas as pd

df = (
    pd.read_parquet("data/measurements.parquet", engine="pyarro
        .groupby("station_name")
        .agg(["min", "mean", "max"])
)
df.columns = df.columns.get_level_values(level=1)
df = df.reset_index()
```

Freedium

```
print("{", end="")
for row in df.itertuples(index=False):
    print(
        f"{row.station_name}={row.min_measurement:.1f}/{row.mea
        end=", "
    )
print("\b\b} ")
```

Dask

Copy

```
import dask.dataframe as dd

df = (
    dd.read_parquet("data/measurements.parquet")
        .groupby("station_name")
        .agg(["min", "mean", "max"])
        .compute()
)

df.columns = df.columns.get_level_values(level=1)
df = df.reset_index()
df.columns = ["station_name", "min_measurement", "mean_measurem
df = df.sort_values("station_name")

print("{", end="")
for row in df.itertuples(index=False):
    print(
        f"{row.station_name}={row.min_measurement:.1f}/{row.mea
        end=", "
    )
print("\b\b} ")
```

Polars

Freedium

```
import polars as pl

df = (
    pl.scan_parquet("data/measurements.parquet")
        .group_by("station_name")
        .agg(
            pl.min("measurement").alias("min_measurement"),
            pl.mean("measurement").alias("mean_measurement"),
            pl.max("measurement").alias("max_measurement")
        )
        .sort("station_name")
        .collect(streaming=True)
)

print("{", end="")
for row in df.iter_rows():
    print(
        f"{row[0]}={row[1]:.1f}/{row[2]:.1f}/{row[3]:.1f}",
        end=", "
    )
print("\b\b} ")
```

DuckDB

Copy

```
import duckdb

with duckdb.connect() as conn:
    data = conn.sql("""
        select
            station_name,
            min(measurement) as min_measurement,
            cast(avg(measurement) as decimal(8, 1)) as mean_mea
            max(measurement) as max_measurement
        from parquet_scan('data/measurements.parquet')
        group by station_name
        order by station_name
    """)
```

```
                f"{row[0]}={row[1]}/{row[2]}/{row[3]}",
                end=", ",
            )
        print("\b\b} ")
```

1BRC with Parquet Results

It looks like we have a clear winner:

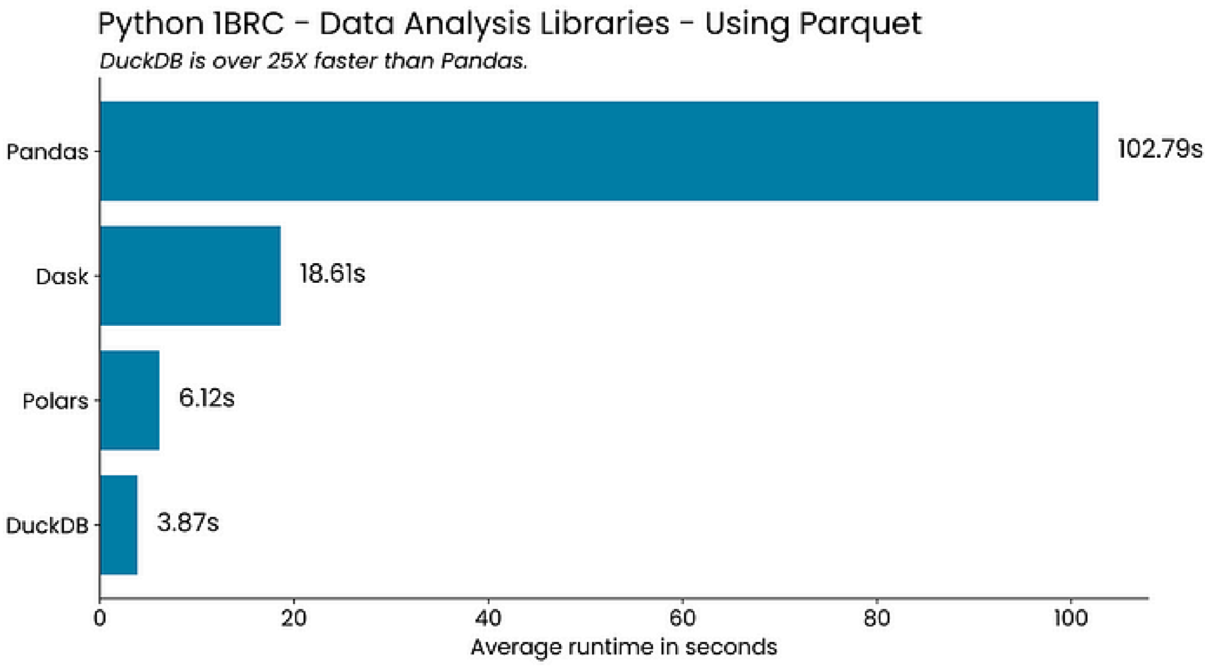


Image 4- Data analysis libraries on Parquet format runtime results (image by author)

The DuckDB implementation on the Parquet file format reduced the runtime to below 4 seconds! It's still about 2.5x times slower than the fastest Java implementation (on `.txt`), but it's something to be happy with.

Conclusion

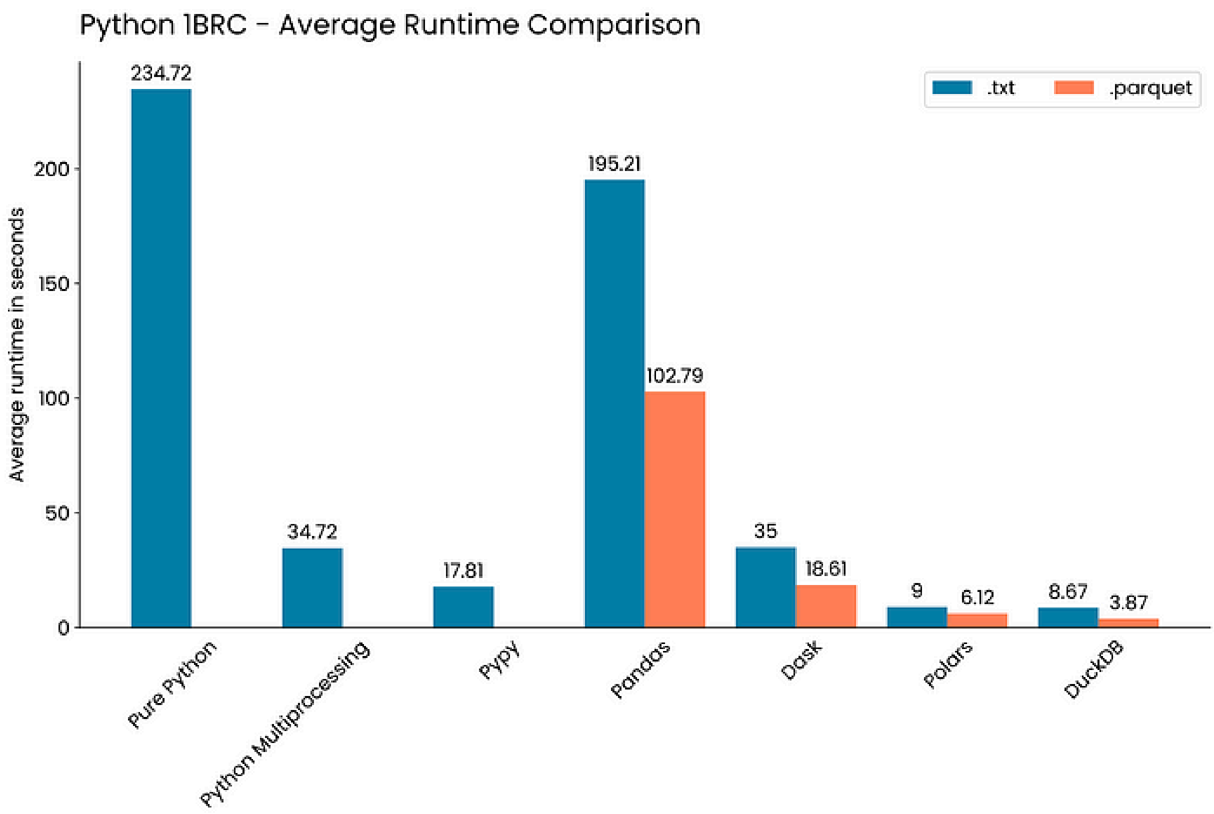


Image 5 — Average runtime results for all approaches (image by author)

Sure, only the first three columns obey the official competition rules, but I don't care. Speed is speed. All is fair in love and Python performance optimization.

Python will never be as fast as Java or any other compiled language — that's the fact. The question you have to answer is *how fast is fast enough*. For me, less than 4 seconds for 1 billion rows of data is well below that margin.

What are your thoughts on the 1 Billion Row Challenge? Did you manage to implement a faster solution? Let me know in the comment section below.

Read next:

Freedium

Rows in 1 Minute

Process huge volumes of data with Python and DuckDB — An AWS S3 example.

towardsdatascience.com

#python #data-science #programming #coding #benchmark