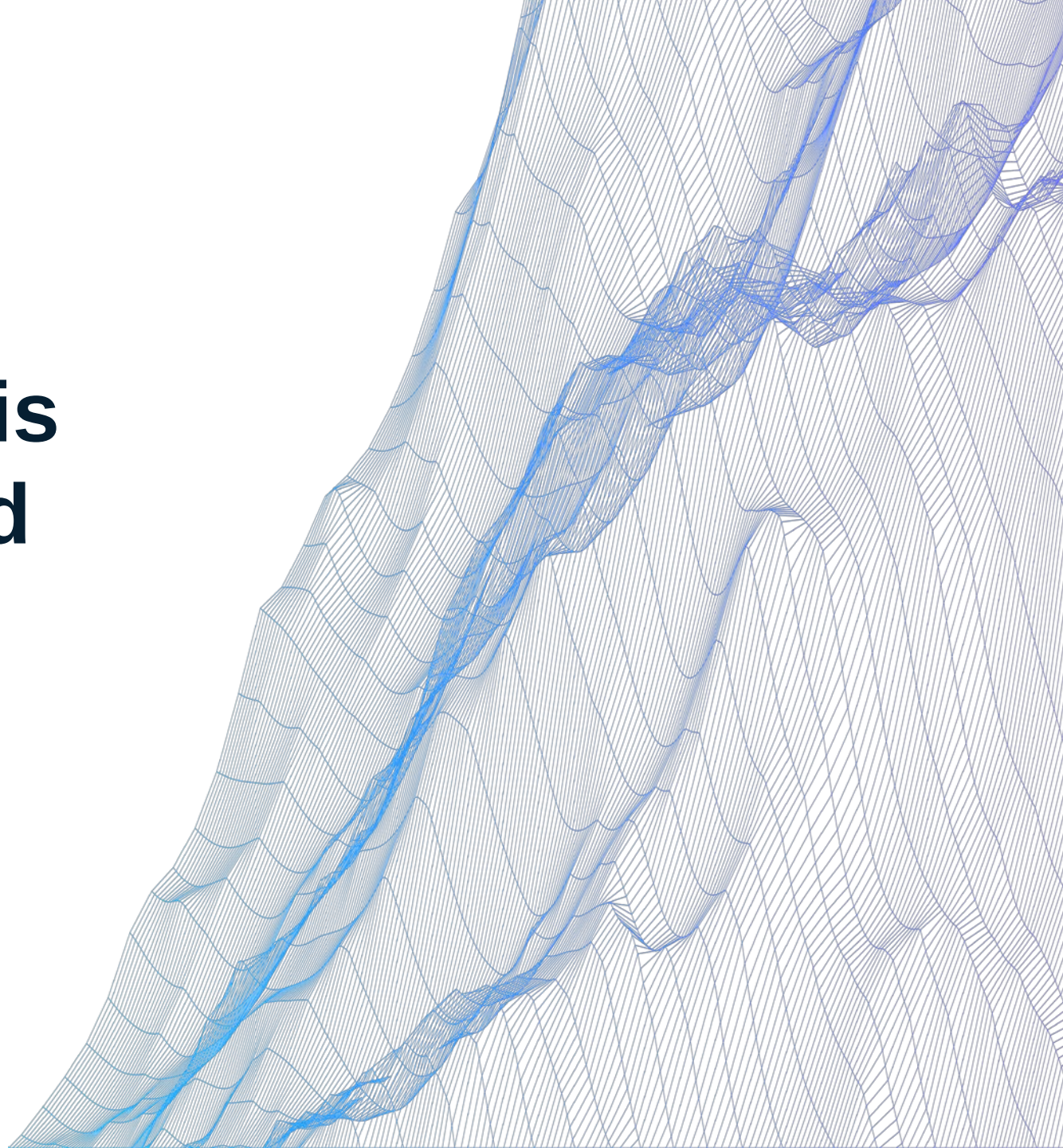
Why your data analysis code runs so slow and what to do about it

Tips and Tricks for economists

Dr. Jannic Alexander Cutura
European Central Bank* & DSTI School of Engineering
jannic.cutura@dsti.institute



*All views are those of the author and do not represent the views of the ECB, ESCB or DSTI.

In my own journey I slowly transitioned from research in financial economics to software/data engineering

Some things I learned in software engineering I wish knew earlier…

[Oct 2014 – June 2020]

MSc./Ph.D. Student
Goethe University

Key activities:
- Conducting research in financial economics
- Teaching & thesis supervision
- Research stays: BIS, IMF & Columbia University

Technologies used:
- Stata
- R
- Excel
- LaTeX

[Jully 2020 – July 2022]

Trainee / Research Analyst
Financial Stability & Monetary Policy

Key activities:
- Data preparation for internal and external reports
- Data quality controls & data pipelines for granular datasets

Technologies used:
- SQL (Hive, Oracle)
- Python & Spark
- Git

[August 2022 – Augst 2024]

Software Engineer
Stress Testing

Key activities:
- Implement risk models based on business specifications
- Implement data quality checks to validate stress test data against benchmark data

Technologies used:
- Python
- AWS
- Git & CI/CD

[September 2024 – today]

Senior Data Engineer
Supervisory Technologies

Key activities:
- Spark ETL migration from Cloudera to AWS Glue
- Migrating from WLA to Apache Airflow

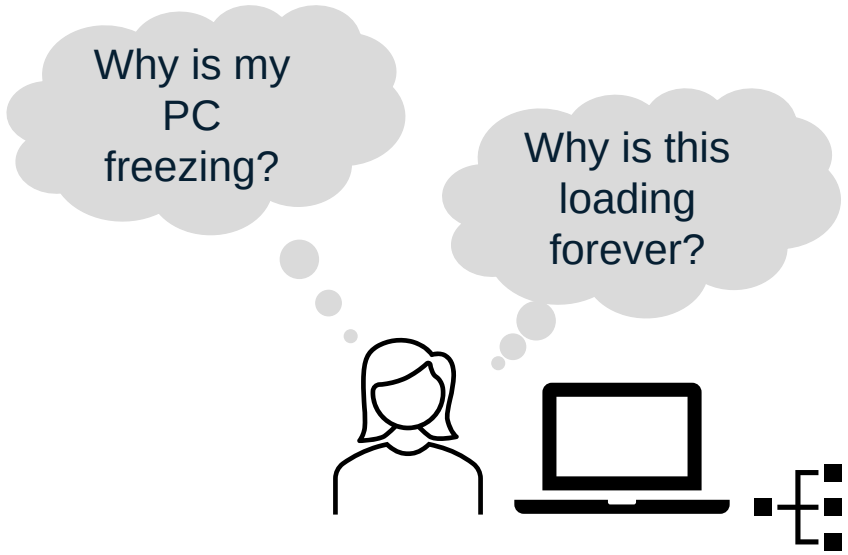
Technologies used:
- Python & Spark
- Terraform & AWS
- Git & CI/CD
- Docker & Kubernetes

…

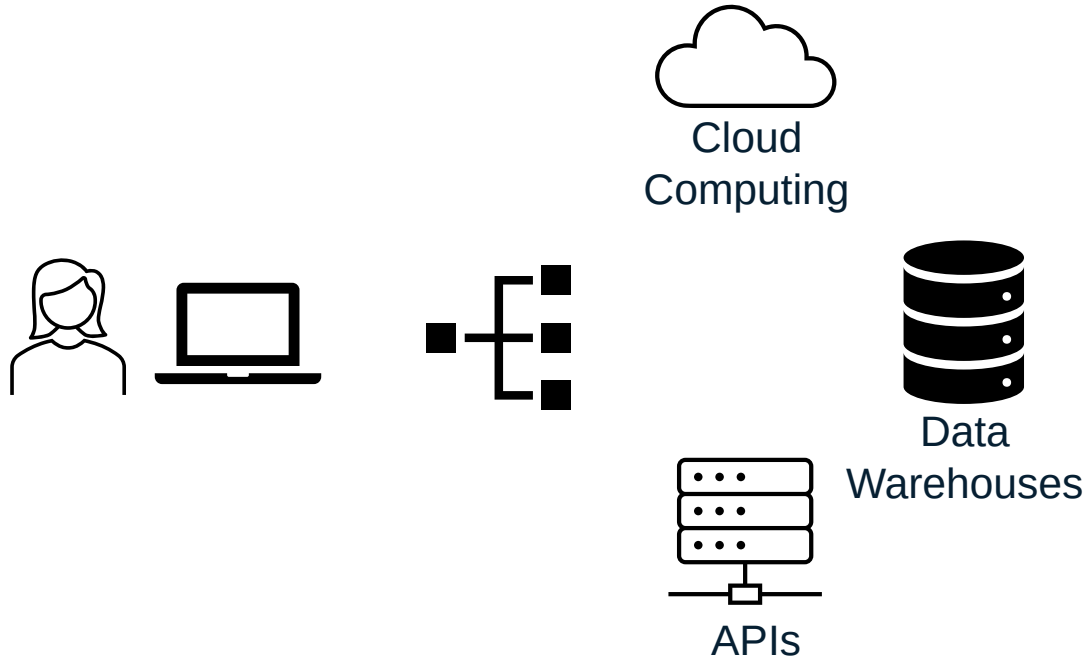
It is imperative to understand what makes your computer slow before we turn to more powerful cloud-based resources

Optimizing code locally will help you make most of more remote alternatives

Crunching numbers *locally*



Crunching numbers *somewhere* else



Crunching numbers locally

01

A PC works by reading data from hard drive/network into memory from where the CPU can take data to make fast operations

Each component can present a bottleneck to slow down your code / data analysis

Data is stored on hard drive or network drive(s)

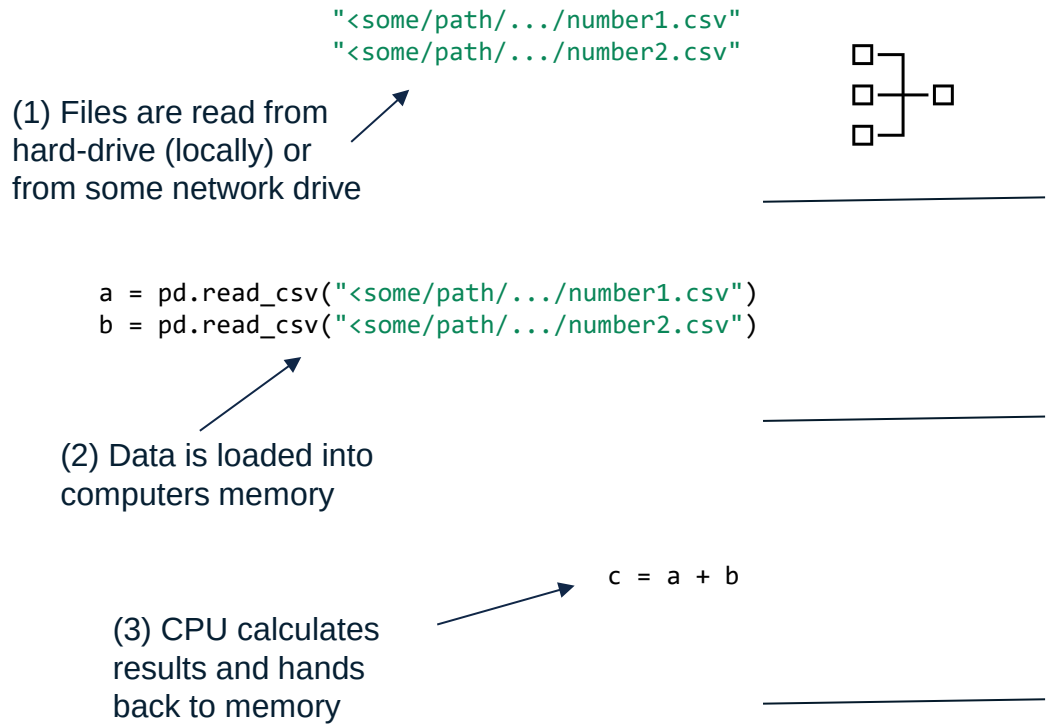
- Data is persisted somewhere (e.g., on Read-Only Memory (ROM), i.e., hard-drive or somewhere on a network). This can be (very, very) slow to read from
- NVMe SSD ~3-7 GB/s sequential vs. HDD: ~100-200 MB/s sequential
- Network storage: Highly variable (10 MB/s to 10+ GB/s depending on infrastructure)

Data is loaded into Random Access Memory (RAM)

- Depending on how much memory you have, this can become a bottleneck if your data files > memory

Central Processing Unit (CPU) performs calculations

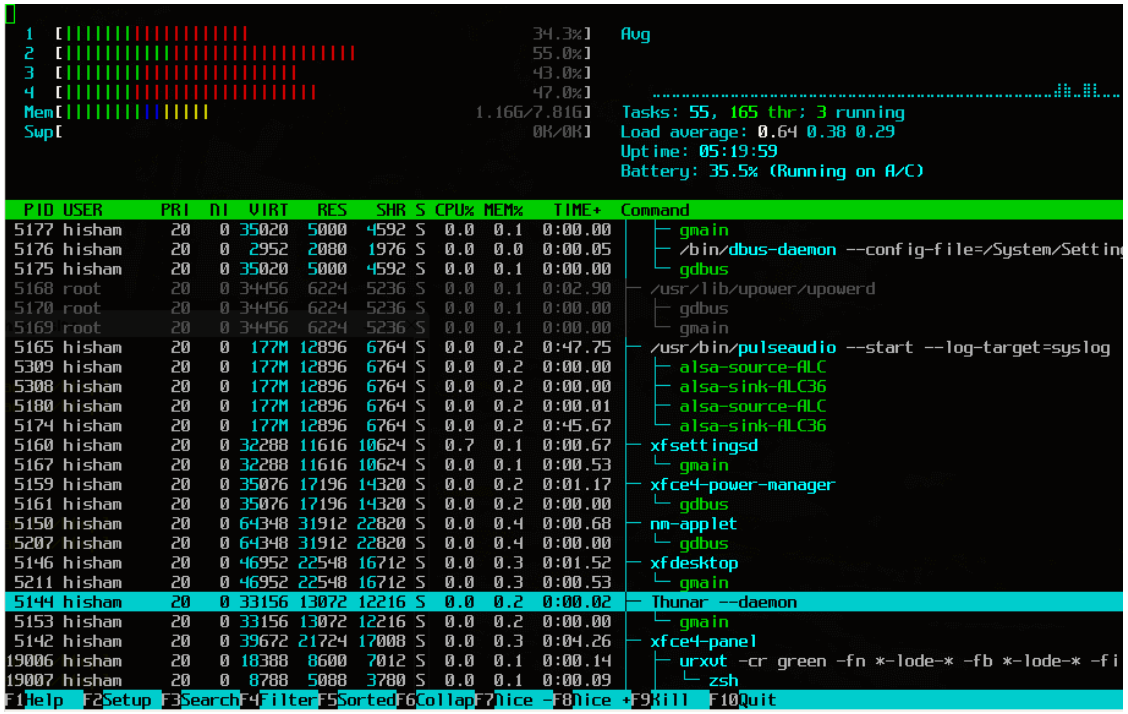
- This can fast, depending on the calculation
- If you can parallelize it well, more cores can significantly speed up the problem
- Side note: for some problems (matrix algebra, aka ML/AI) GPU performs better



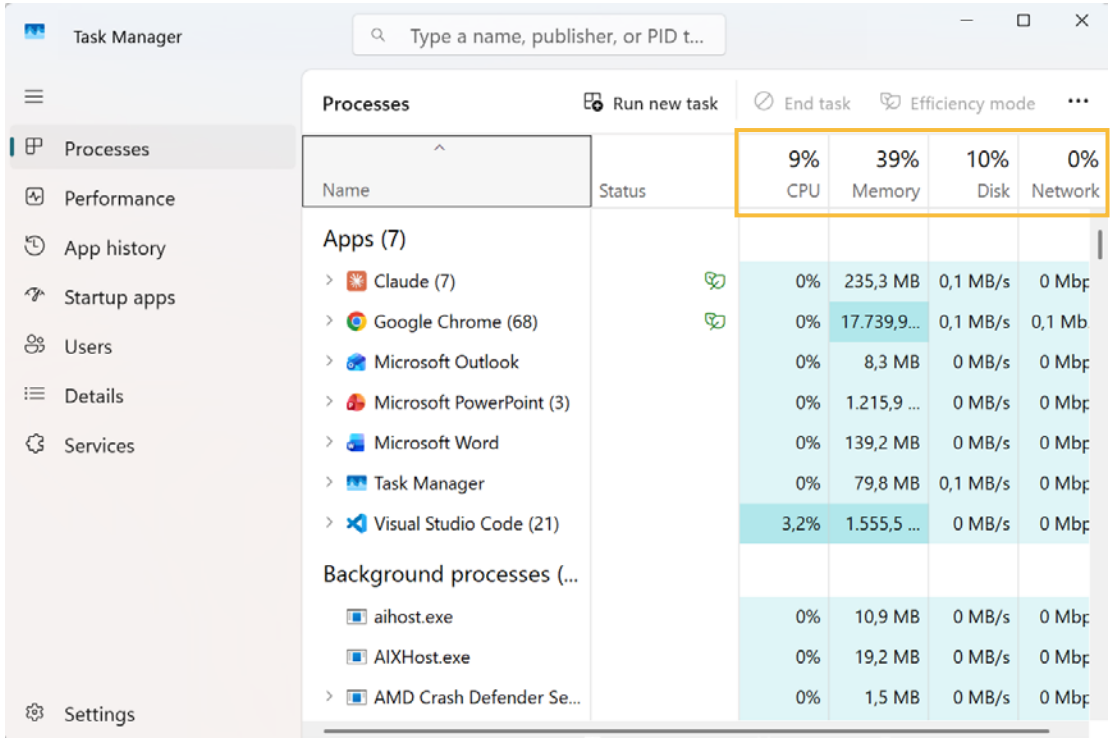
You can check this yourself next time your system is slow…

You will see either CPU, memory or I/O be maxing out…

`htop` command in linux



Windows Task Manager



Source: Author's screenshot (right) and https://www.tecmint.com/wp-content/uploads/2012/08/Htop-Linux-Process-Monitoring-Tool.png (left)

Three bottlenecks when analyzing data: poor network performance, time complexity and space complexity

(Poor) Network Performance

- When dealing with large files stored on a network drive, the speed at which data can be transferred to your local machine is heavily dependent on the network's performance. If the network is congested or the connection is slow, it can create a significant bottleneck in the data pipeline, causing delays in data retrieval.
- This lag can be especially frustrating when you need to frequently access or stream large datasets, as each request might take much longer than anticipated. The result is a workflow hampered by extended wait times, reducing overall productivity and potentially leading to inconsistent or incomplete data being loaded for analysis.

Space complexity

- Large datasets often exceed the available memory (RAM) on a typical computer, leading to critical performance issues. When your machine attempts to load a dataset that is too large for the available memory, it resorts to using disk space as an extension of RAM, a process known as "paging" or "swapping“.
- This operation is considerably slower than using RAM alone, causing the system to become sluggish. In extreme cases, the machine may freeze or crash. Even when the system remains operational, the time required to load and manipulate the data increases dramatically.

Time complexity

- The number of cores in your computer determines its ability to perform parallel processing, which is essential for handling computationally intensive tasks, especially with large datasets. When you have a limited number of cores, your machine can only perform a certain number of calculations simultaneously.
- This limitation becomes evident as the dataset grows larger, requiring more computational power to process in a reasonable time. This bottleneck can be particularly problematic when dealing with complex algorithms or when trying to scale your analysis to meet the demands of big data.

Understanding which problem limits your data analysis is key to improve upon

Time complexity is about *CPU-bound* problems, space complexity is about *memory-bound* problems

Understanding which one your code suffers from is key to make it better

Time complexity Problem

- Consider the Fibonacci sequence:

$$F(n) = F(n-1) + F(n-2) \rightarrow 0,1,1,2,3,5,8,13,21,?$$

- Space complexity?
  - Step 1: 0,1 $\rightarrow 0 + 1 = 1$, carry result & 1, forget 0
  - Step 2: 1,1 $\rightarrow 1 + 1 = 2$, carry result & 2nd 1, forget 1st 1
  - Step 3: 1,2 $\rightarrow 1 + 2 = 3$, carry 2, forget 1st 1
  - Step 4: 2,3 $\rightarrow$ ...
  - Need only ever two numbers in memory $\rightarrow O(1)$ "constant space complexity"
- Time complexity? Need n calculations to get the *n-th* number
- Linear time complexity: "I cannot split up the work" $O(n)$ time complexity
- No amount of compute helps, only math helps: $F(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$

Space complexity Problem

- $100{,}000 \times 100{,}000$ matrix of zeros
- Add 1 to each cell. What is the result?
- $100{,}000 \times 100{,}000$ matrix of ones
- Let python do it:

```
import numpy as np
N=100_000
zeros = np.zeros((N, N))
```

- Oops:

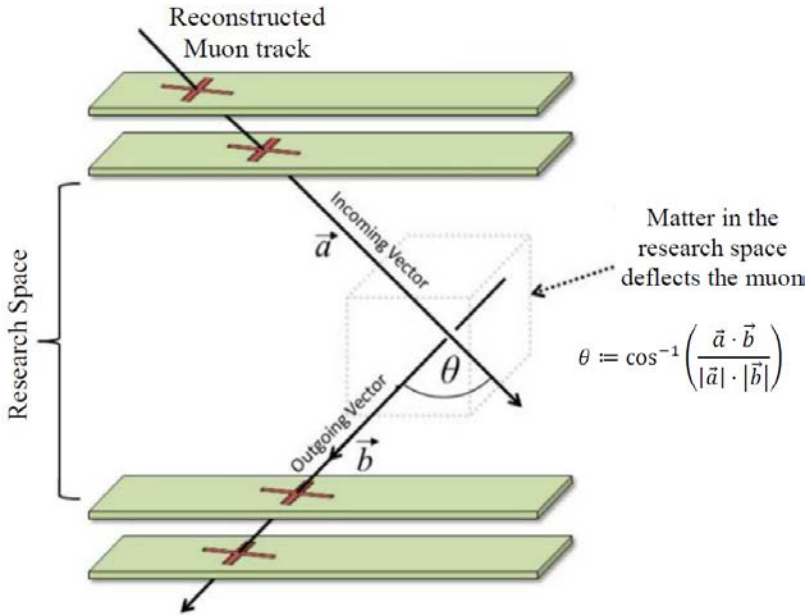
```
>> MemoryError: Unable to allocate 74.% GiB for an array with shape (100000, 100000) and data type float
```

- We would need 75GB of available memory to populate the matrix
  - Note that we have not even computed anything (no the "+1" part yet), just initializing the matrix

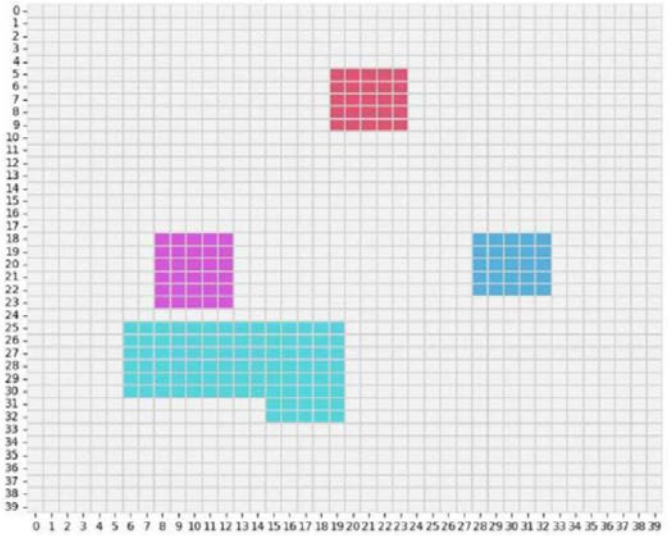
Footnote: 1. If you implemented Fibonacci recursively, the space complexity is $O(n)$ and the time complexity is $O(2^n)$.

Case study Proper Data Typing: Muon tomography data science challenge

Stylized depiction of muon tomography experiments



Ground truth of materials in research space of same example



Median of cosine similarity by cell of one example

Footnote: 1. One million rows of four sets of (x, y, z) coordinates with three decimal points precision stored as float16 vs int64.

Source: [left panel] Adapted from Lloret, Lara, del Arbot Pablo: Deep convolutional neutral networks applied to muon tomography images, 2019, [middle/right panel] Authors simulation. You can find the full paper here: https://ieeexplore.ieee.org/document/10386932

Provided data was stored very inefficiently as 60GB worth of CSV files

Original data contains (x, y, z) coordinates, each have “three-digit precision”, and many zeros, are interpreted as floats (float64 in python) by most languages:

x1	y1	z1	x2
0.123000000000000	0.294000000000	0.454000000000	...
-0.153000000000000	0.654000000000	0.674000000000	...
...	...	...	...

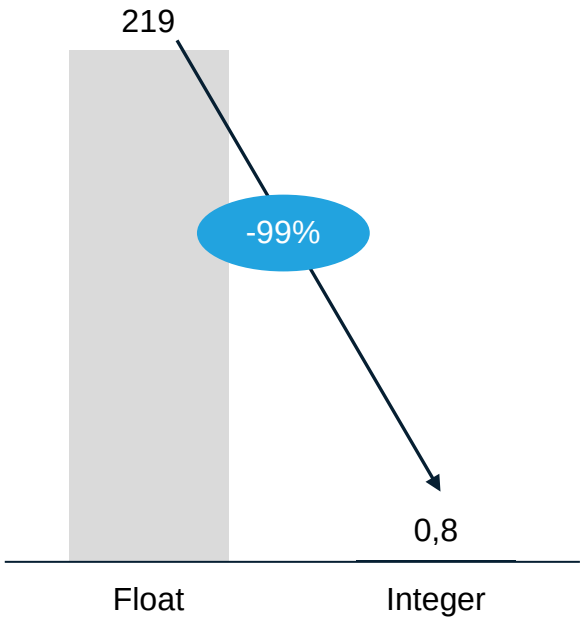
Multiplying by 1000 and storing as int(16) allows to store the data much more efficiently without losing information:

x1	y1	z1	x2
123	294	454	...
-153	654	674	...
...	...	...	...

Formatting your data to the applicable data type can save tremendous amount of space & memory

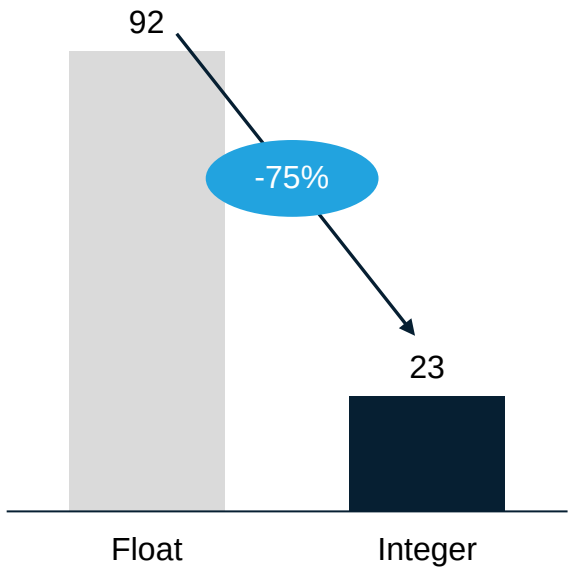
200x disk space savings

File size,
Megabyte



4x Memory savings

Memory usage,
Megabyte



Lessons learned

- Critically reflect on your data: not just conceptually, but also in terms of data types.
- Can you use less memory/space greedy types without losing information? Avoid decimal types (e.g. use 1,234,456EUR, not 1.23456 MEUR).
- Can you afford to lose (some) information?
- Use built-in functions where possible, e.g. Stata's `conserve`
- Each language has their own types, research it and choose the suitable one if you have large tables
- Only do this is memory/disk space is becoming a bottleneck.

Footnote: 1. Simulated data of one million rows of four sets of (x, y, z) coordinates with three decimal points precision stored as float64 vs. int16.

Source: Authors simulation, replication material can be obtained at https://github.com/JannicCutura/bplim-2025

Let’s see what the computer sees under the hood

Level 1: The Physical Reality

The Basic Unit: **A Bit**

- A bit is the smallest unit of information
- Physically, it's a tiny capacitor that either:
  - **Has charge** (high voltage, ~3-5 volts) → we call this 1 or "ON“
  - **Has no charge** (low voltage, ~0 volts) → we call this 0 or "OFF"

Think of it like a light switch: either on or off. That's it. Your entire computer is built from billions of these switches.

Level 2: From Voltage to Binary Numbers

Since we only have 1s and 0s to work with, we express numbers in base 2: $123 \equiv 1\mathbf{1}11\mathbf{0}1\mathbf{1} = 1 \times 2^6 + \mathbf{1} \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + \mathbf{0} \times 2^2 + 1 \times 2^1 + \mathbf{1} \times 2^0$

```
Decimal    Binary    What the computer "sees"
-------    ------    ------------------------
0          0         [no voltage]
1          1         [voltage]
2          10        [voltage][no voltage]
3          11        [voltage][voltage]
4          100       [voltage][no voltage][no voltage]
5          101       [voltage][no voltage][voltage]
123        1111011   [voltage][voltage][voltage][voltage][no voltage][voltage][voltage]
```

Storing as integer we can save considerable space

Option 2: Store as int16 (2bytes)

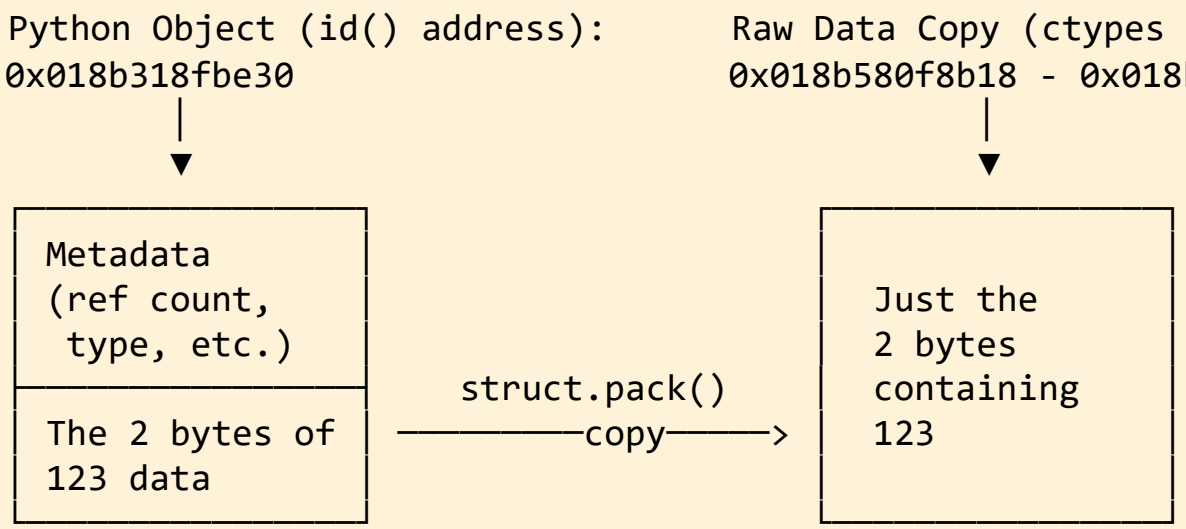
```
In memory: [01111011][00000000]
           └──────── 16 bits ─────┘

a = 123
------------------------------------------------------------------------
Python's id(a) function
------------------------------------------------------------------------
  id(value) = 140724953211496
  In hex:     0x00007ffd14d8e268

------------------------------------------------------------------------
ctypes - Raw byte memory addresses (as int16)
------------------------------------------------------------------------

  The 123 as int16 bytes are stored at:
    Byte 0: Address 0x02447e8a5e98 = [01111011] = 123
    Byte 1: Address 0x02447e8a5e99 = [00000000] =   0
```

Footnote: The numbers ranged from $[-0.3{,}00\ 0.300]$ or $[-300{,}300]$. Hence $int8 = x \in [-128, 127]$ is not enough, we need $int16 = x \in [-32768, 32767]$.

Small detour: Object construction in python

Ignore this slide if you find it confusing!

Option 2: Store as int16 (2bytes)



Footnote: The numbers ranged from $[-0.3{,}00\ 0.300]$ or $[-300{,}300]$. Hence $int8 = x \in [-128, 127]$ is not enough, we need $int16 = x \in [-32768, 32767]$.

We can store the data as floating point, but that is expansive:

Option 1: Store as float64 (8bytes, IEEE754 standard): $value = (-1)^S \times 1.F \times 2^{E-1023}$

```
In memory: [00111111][11001101][01110000][10100011][11010111][00001010][00111101][01110001]
           └──────────────────────────────── 64 bits ──────────────────────────────────────┘

0.123 in binary scientific notation ≈   1.9680000543959... × 2^(1019-1023)
                                          ↑                          ↑
                                       mantissa                   exponent
Stored as:
[0][01111111011][1111101111100011011001100110011001100110011001101101101]
 +      1019                        1.968
 ↑       ↑                           ↓
sign  exponent              52 bits of mantissa
     (11 bits)

1.9680000543959... × 2^(1019-1023) = 0.122999999999999998223643160599749535322189331054687 5
```

Footnote: The mantissa is calculated as $1 + (1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-3} + 1 \times 2^{-4} + 0 \times 2^{-5} + \cdots) = 1.968$

Avoid unnecessary copies of your data in memory, but be aware of in-place modifications

```
df_original = pd.DataFrame({
    'A': [1, 2],
    'B': [10, 20],
    'C': [100, 200]
})
df_new = df_original
df_new.loc[0, 'A'] = 999

print(df_new)

     A   B    C
0  999  10  100
1    2  20  200

print(df_original)

     A   B    C
0  999  10  100
1    2  20  200
```

Lessons learned

- As you code be aware of assignment vs. copying of data
- Both have their place:
  - A copy allows independent modification but eats up memory
  - An assignment is lightweight, but be aware of in place modifications
- Run `memory_reference_demo/demo.py`

Loop accumulation trap occurs when data is copied (under the hood) during a loop

```python
# O(n²): Each concat copies ALL previous data
results = pd.DataFrame()
for file in files:
    data = pd.read_csv(file)
    results = pd.concat([results, data])

# O(n): Append references, concat once
dfs = []
for file in files:
    data = pd.read_csv(file)
    dfs.append(data)

results = pd.concat(dfs)
```

Files	Bad Approach	Good Approach	Speedup
20	210 copies	1 copy	210x
100	5,050 copies	1 copy	5,050x
250	31,375 copies	1 copy	31,375x

Lessons learned

- `pd.concat()` always allocates new memory and copies all input DataFrames
- In a loop, each iteration copies more data than the previous one (1 + 2 + 3 + ... + n)
- This creates $O(n^2)$ quadratic complexity - devastating for large datasets
- List append is O(1) - only stores an 8-byte memory reference, no data copying
- One final concat is O(n) - copies each DataFrame exactly once
- **Applies to any iterative data collection**: API calls, database queries, file processing
- Run loop_accumulation_pattern/demo.py and /other_accumulation_patterns.py

Most languages offer *vectorized* versions of most operations which will run much, much faster

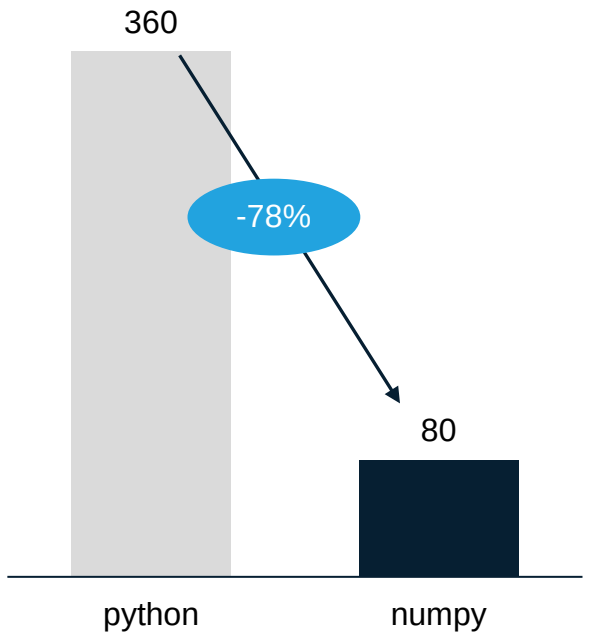
Vectorized operations “push application logic down” to machine code

200x disk space savings

```python
n = 10_000_000

# pure python
python_list = list(range(n))
result_python = sum(python_list)

# vectorized numpy
import numpy as np
numpy_array = np.arange(n)
result_numpy = np.sum(numpy_array)
```

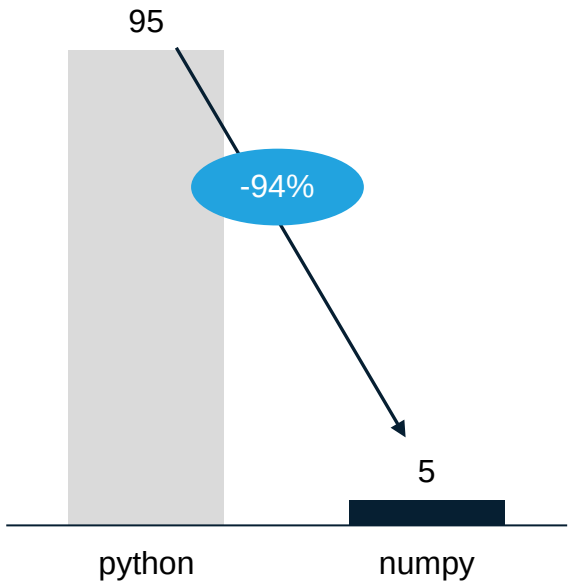
4.5x Memory savings

Memory footprint,
Megabyte



20x times faster summation

Execution time,
milliseconds

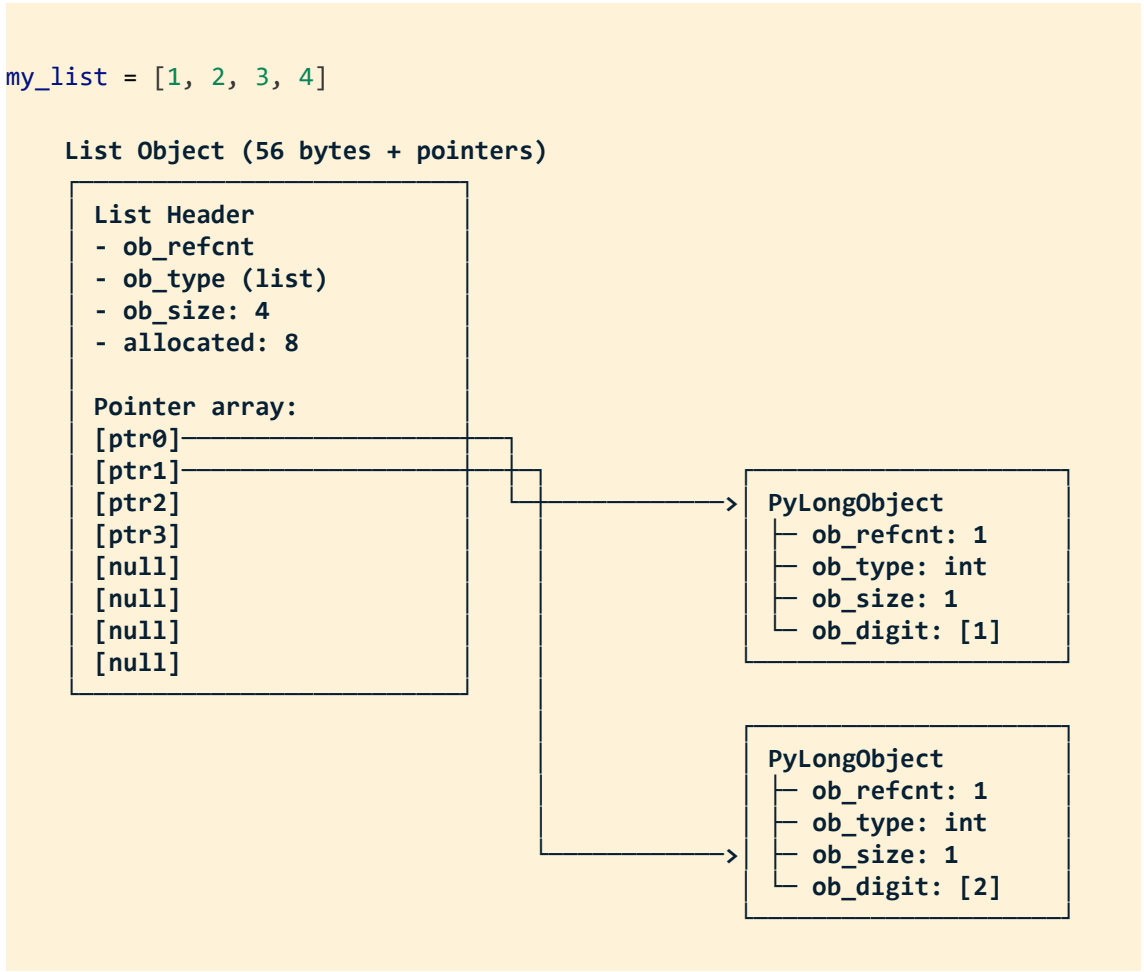


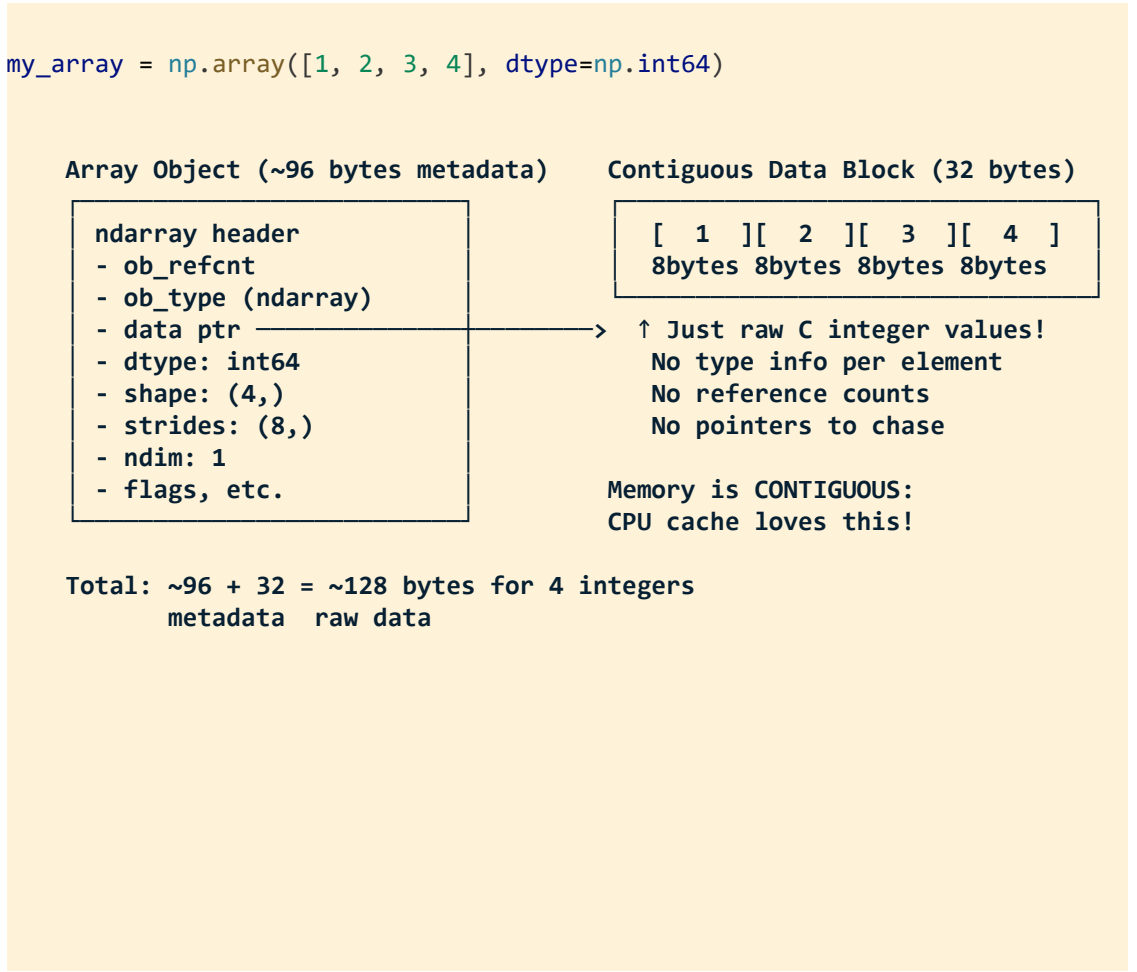
Footnote: 1. Simulated data of ten million numbers using pure python or numpy

Source: Authors simulation, replication material can be obtained at https://github.com/JannicCutura/bplim-2025

Scattered Python objects waste memory and kill CPU cache performance

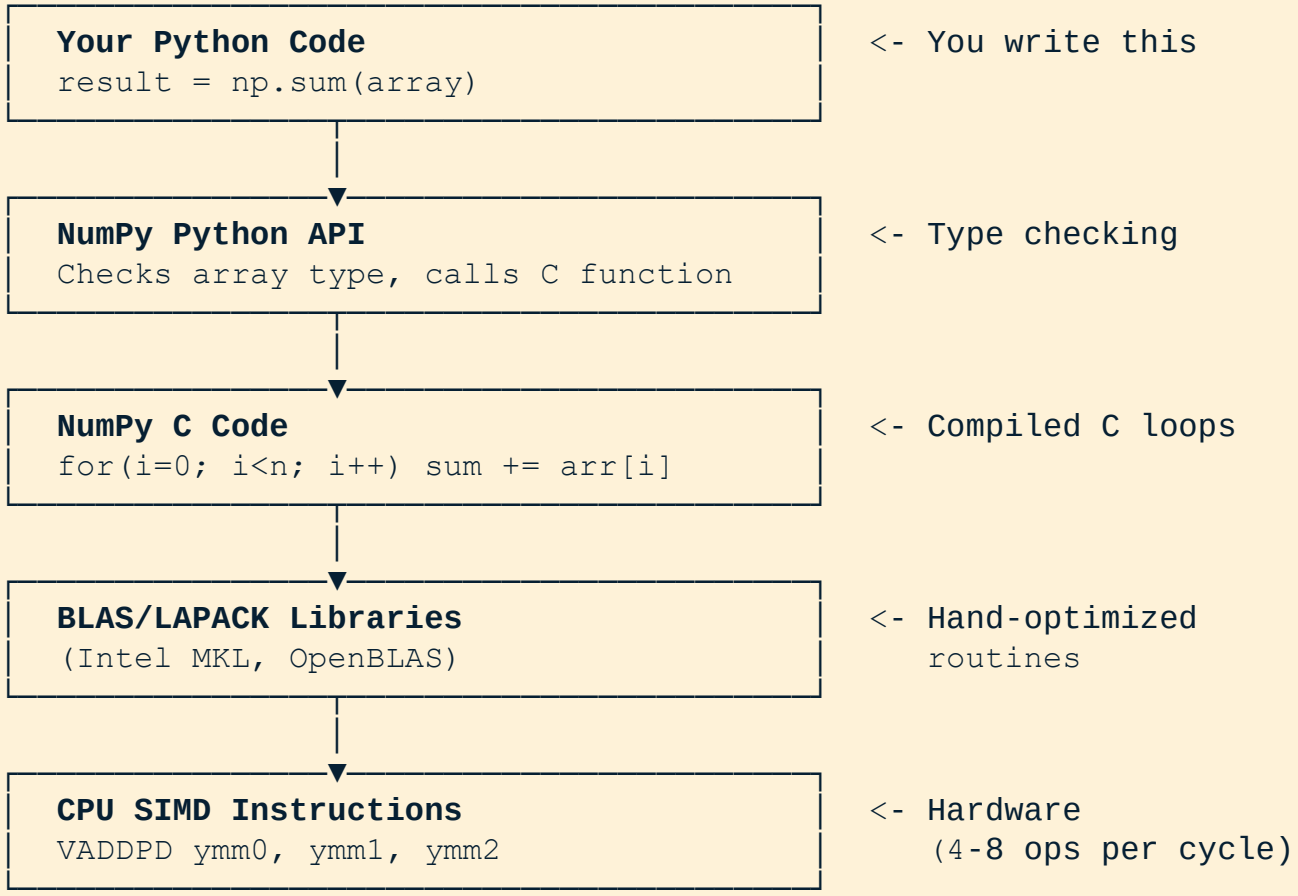
Contiguous memory layout delivers 3.7x memory savings and faster CPU cache performance





Each Python operation climbs the abstraction ladder; NumPy stays at the hardware layer

Vectorization enabled processing 4-8 numbers per CPU cycle instead of one at a time



Lessons learned

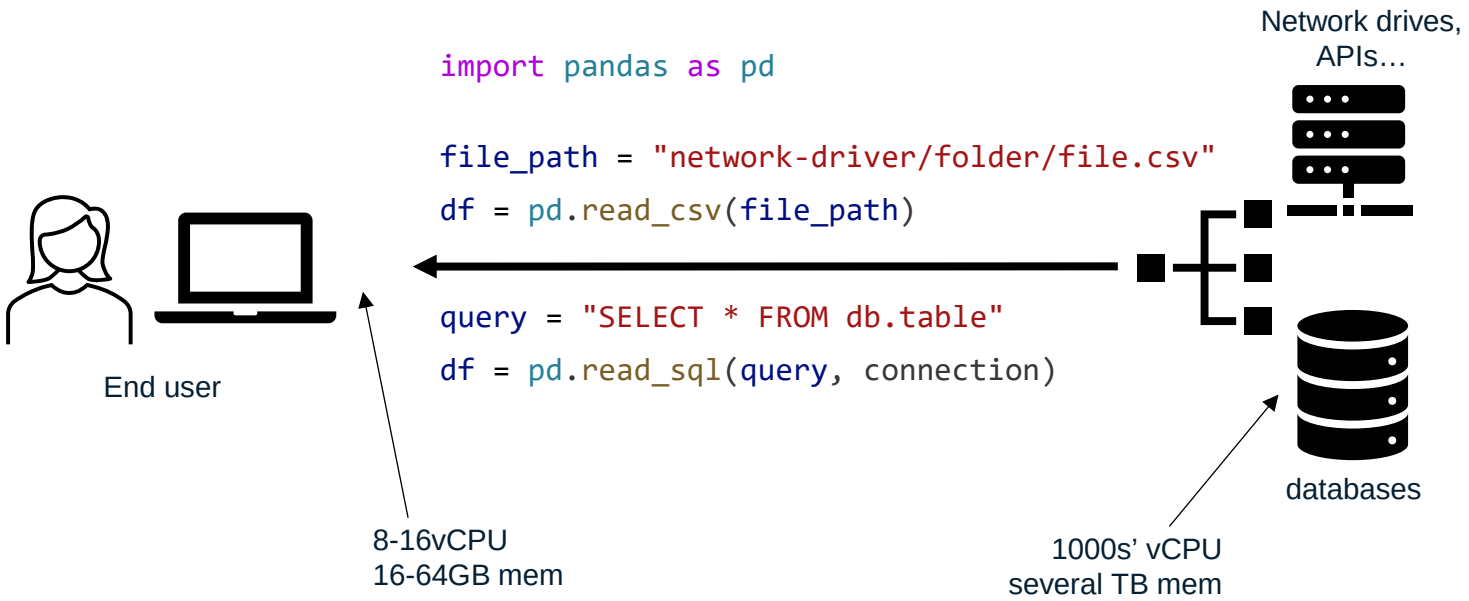
- Most APIs (numpy, polars,…) code looks very clean, like pure python
- Type checking in python and subsequent handover to C
- Compiled C – no more python interpreter overhead!
- Hardware optimized routines
- CPU Single Instruction, Multiple Data (SIMD) instructions allow you to process 4-8 numbers simultaneously

Source: xx

As you start your data analysis journey you are typically loading data from some source to your computer

The source are typically databases, network drives, APIs…

Stylized data retrieval



Data resides on remote sources

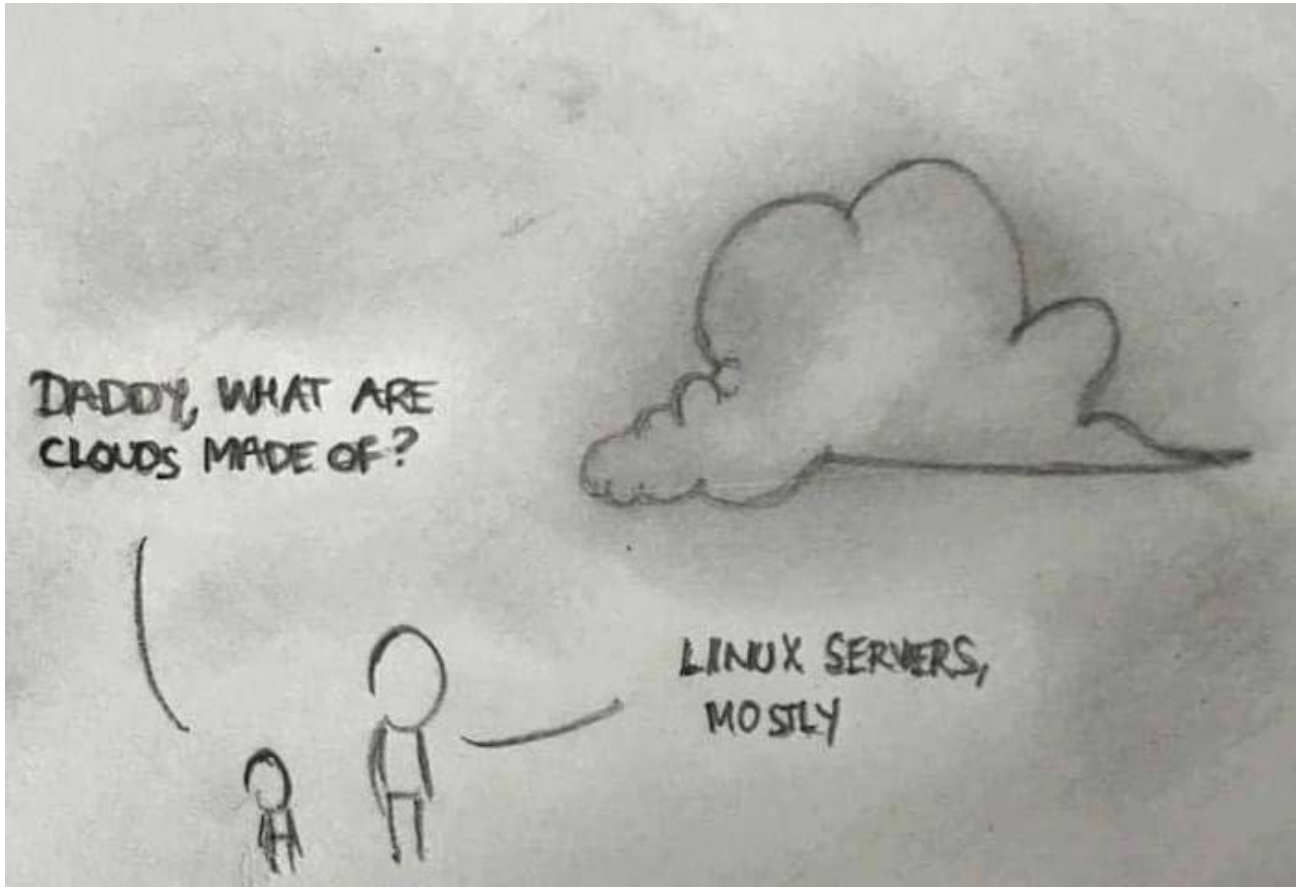
- Data analysts / researches load data to their computer from some (remote) source system into their computer's memory
- Files stored on network drives
- Traditional data warehouses system, accessed using SQL
- Data lakes, which are basically file based systems with a "table format" on top of it
- This yields three bottlenecks:
  - Data travels very slowly across the network
  - Your laptop has limited memory
  - Your laptop has limited vCPU

Crunching numbers *somewhere* else

02

Somewhere else can be anything that offers compute, not just storage

We will focus mostly on data warehouses & data lakes given their popularity

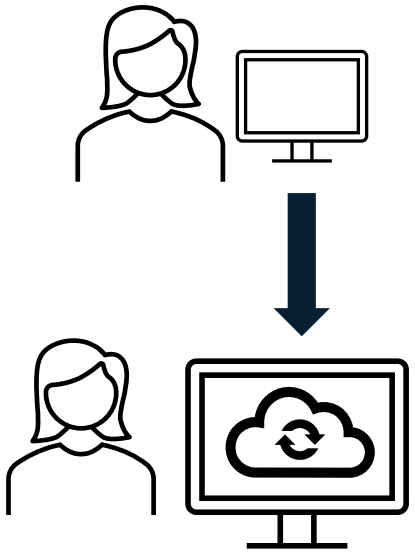


Source: CyberCity; https://www.cyberciti.biz/media/new/cms/2017/05/what-are-cloud-in-cloud-computing-is-made-of-dad.jpeg

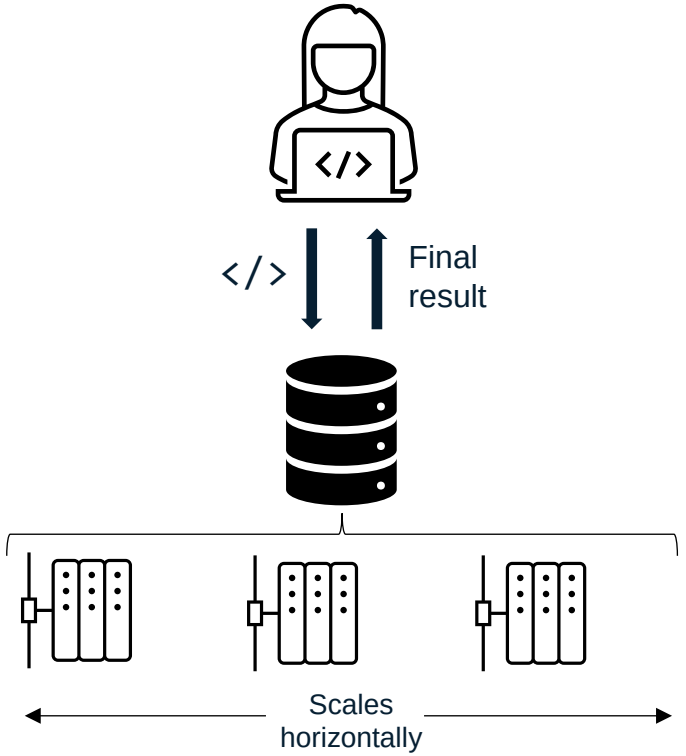
The strategy is to use systems that scale horizontally, not vertically

Vertical scaling is expansive and has technical limits

Vertical Scaling means increasing power of one device



Horizontal Scaling means using power of many devices



Lessons learned

- No hardware ceiling: Vertical scaling hits physical limits of single machines; horizontal scales indefinitely by adding nodes.
- Cost efficiency: Commodity hardware scales cheaper per unit of compute than enterprise-grade vertical upgrades; easier to right-size resources.
- Fault isolation: Node failure in horizontal setup impacts only that node's data; vertical failure takes entire system down.

Move the computation to the data, rather than moving the data to the computation

Read the slide title again.

Bring data to the computation vs…

```
import pandas as pd
df = pd.read_sql("SELECT * FROM orders",
conn)

df = df[df['status'] == 'completed']

results = df.groupby('customer_id')
    .agg({'amount': 'sum', 'order_id':
'count'})
    .reset_index()
    .rename(columns={'amount':
'total_spent', 'order_id': 'order_count'})
)
result['avg_order_value'] =
result['total_spent'] /
result['order_count']
```

… bringing computation to the data

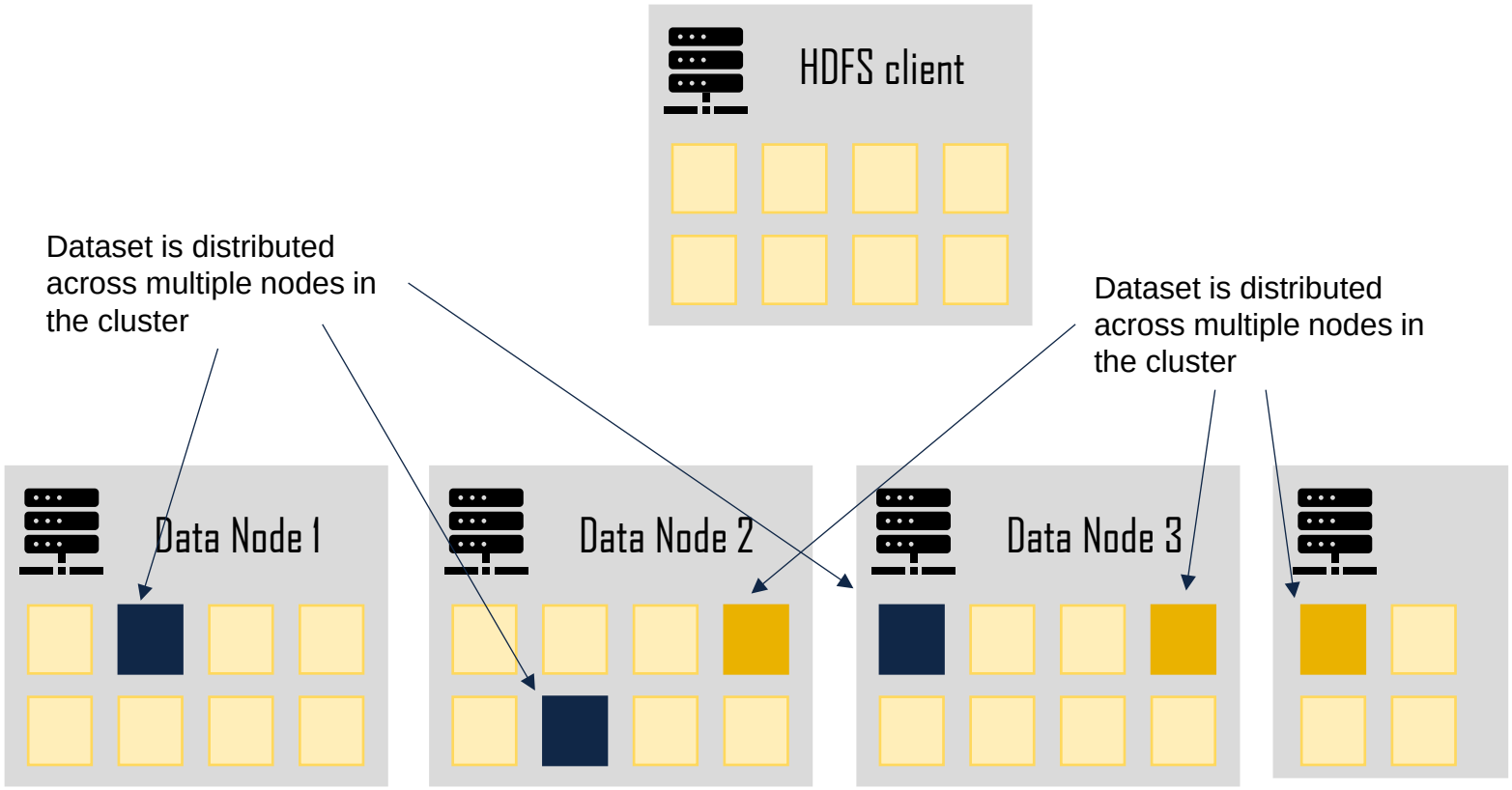
```
import pandas as pd
result = pd.read_sql("""
SELECT
    customer_id,
    SUM(amount) AS total_spent,
    COUNT(order_id) AS order_count,
    SUM(amount) / COUNT(order_id) AS
avg_order_value
FROM orders
WHERE status = 'completed'
GROUP BY customer_id
""", conn)
```

Code what to do, not how to do it

- Declarative programming: Code what you need
- Imperative programming: Code what you want the computer to do
- Declaration optimizes: Spark/SQL states what result you need; query optimizer handles how. Pure Stata/R/Python forces you to specify how, locking you into suboptimal execution paths.
- Optimizer intelligence compounds: Modern compute engines (Iceberg, Spark SQL, DuckDB) apply decades of distributed execution research; Pure Stata/R/Python processing forces you to rediscover those optimizations manually.

In a data lake, data is spread across multiple compute nodes and replicated across the nodes for resilience

Suppose we want to calculate the mean of a dataset too large to fit into our computer's memory…

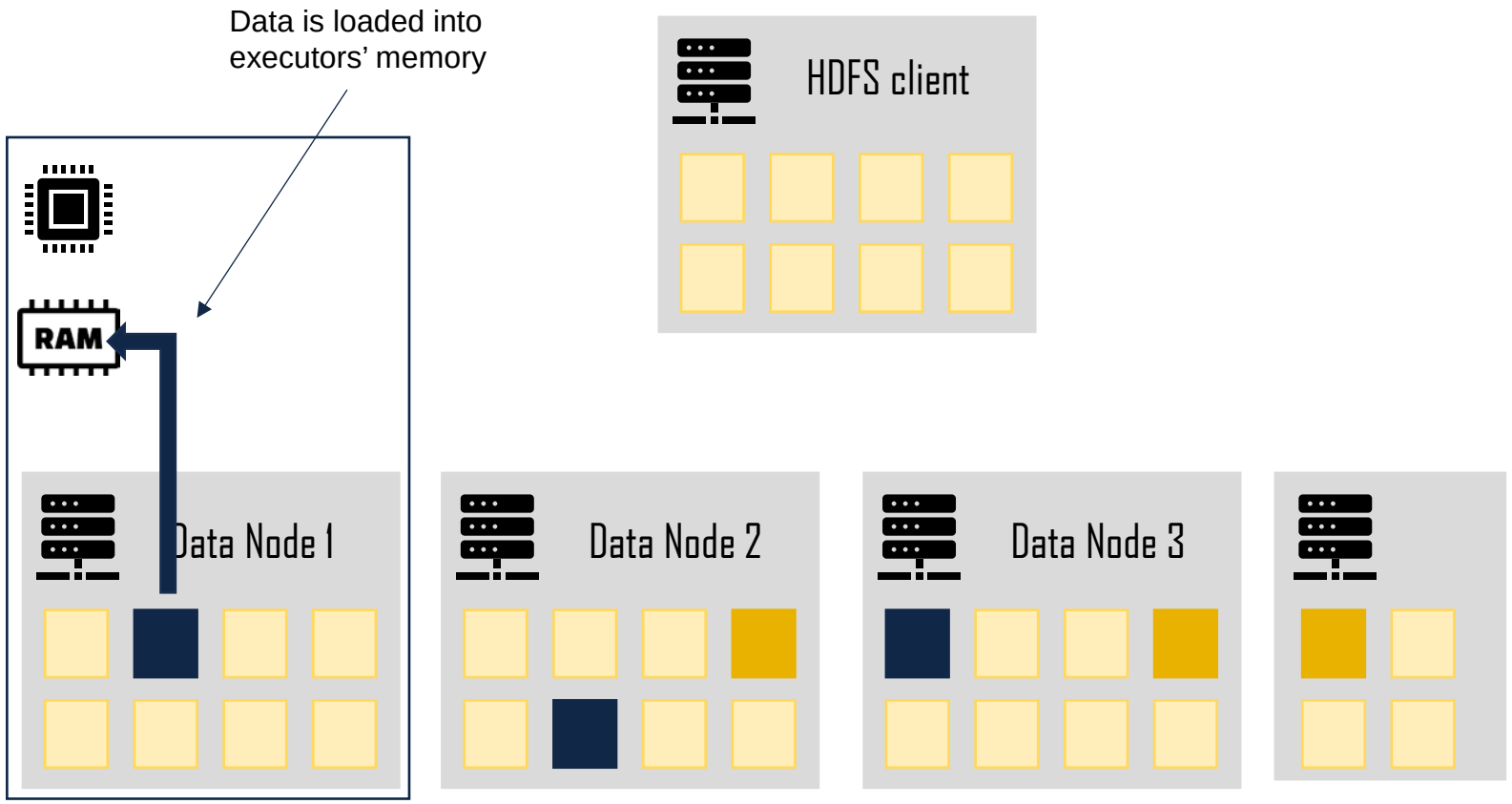


Parallel Calculation

- Consider we have 75GB dataset of numbers which we want to calculate the sum of.
- The data is scattered across the cluster, and typically replicated, such that if any single data nodes goes down, the data is never lost.
- The HDFS client manages the meta data (which files is where).

The strategy in distributed computing is to break up the computation and spread it across multiple workers

Chunks of a large dataset will fit into a given executors' memory

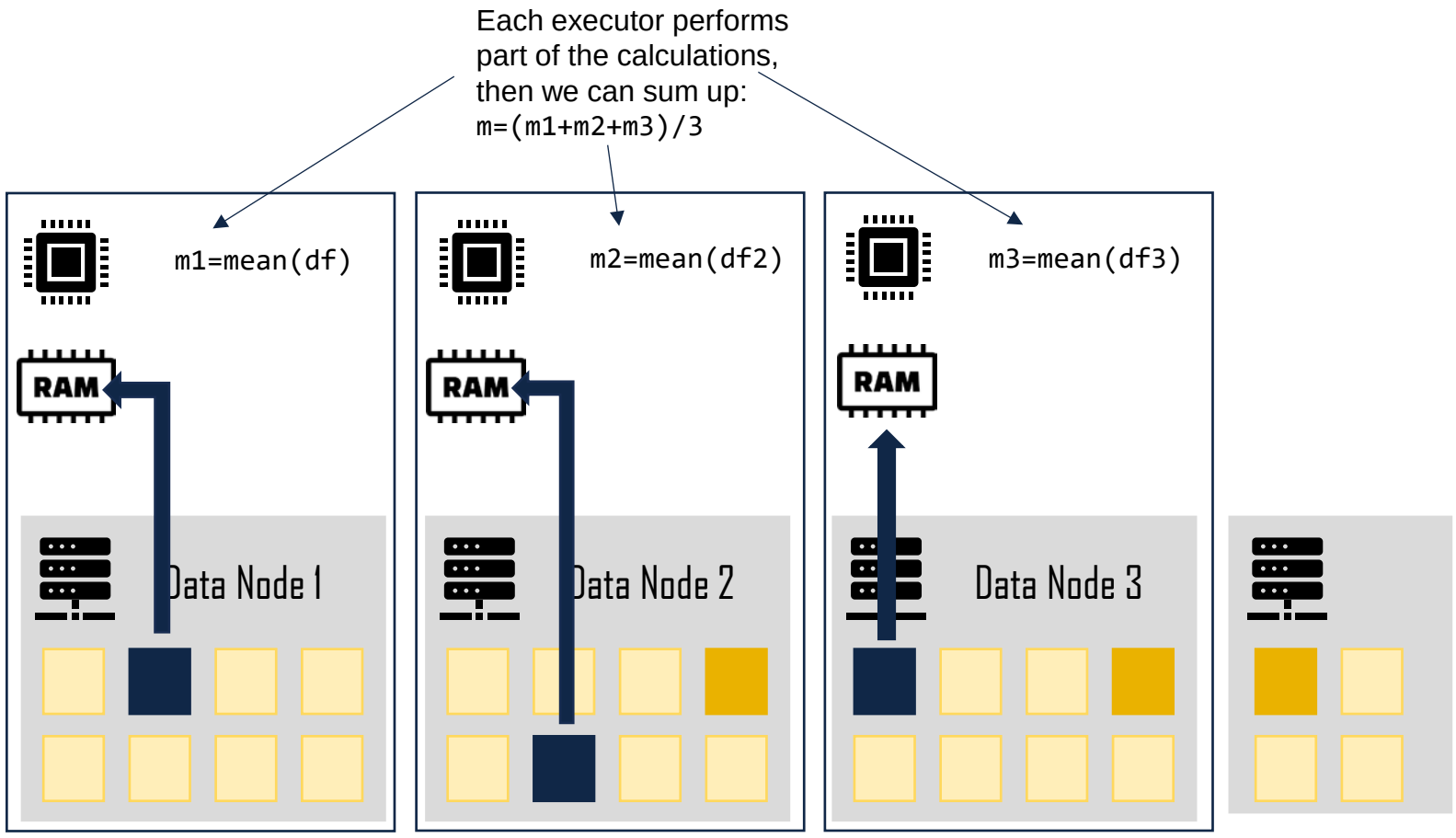


Each chunk fits into that executors memory

- In a Hadoop cluster, data can be loaded directly into the executors' memory to performance computations without having to move data through the network (at least in the initial non-exchanged) data phase.
- "Second generation" object storage-based spark cluster (e.g. S3) do not quite have that, but given the high bandwith inside the data center network still basically benefit from the same feature

By distributing the computation, we can solve the “too much data” problem

By aggregating up the partial results



Results can be aggregated

- Each nodes can calculate the sum of the partial data. Finally, the partial sums can be summed up.
- Problems solved:
  - **I/O:** Takes little time to read since is on the machine's hard drive
  - **Space complexity**: Small chunks fit into executors memory
  - **Time complexity**: Can be cut in three (or however many nodes you have)

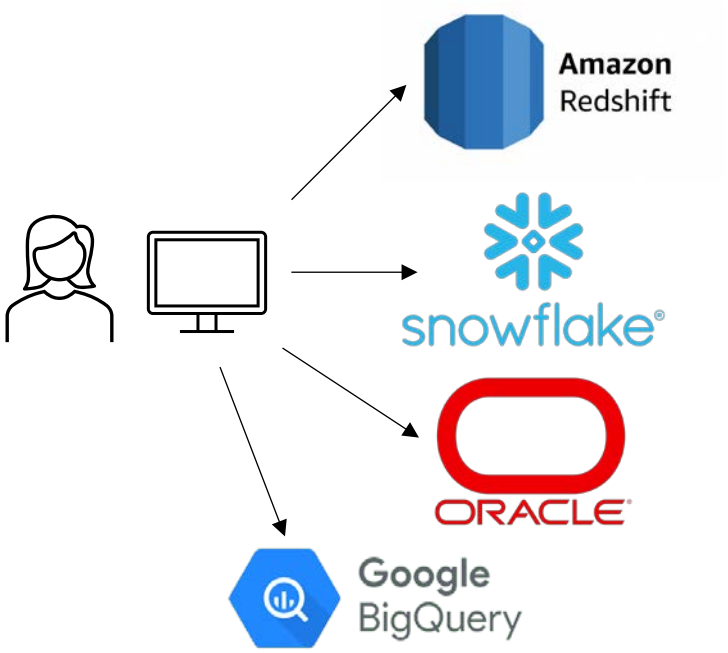
Distributed data processing solve the three classical challenges in data processing

	Challenges when processing locally	How spark solves this
1	I/O limitations slow network speed limited space on local hard drive	Less network travel involved; data is either on the node or close by some cloud object storage
2	Limited memory to store data on retail grade hardware	Since it is horizontally scale-able we have basically unlimited memory available (if you can afford it)
3	Small number of CPU cores limits parallel execution	Since it is horizontally scale-able we have basically an unlimited number of cores available (if you can afford it)

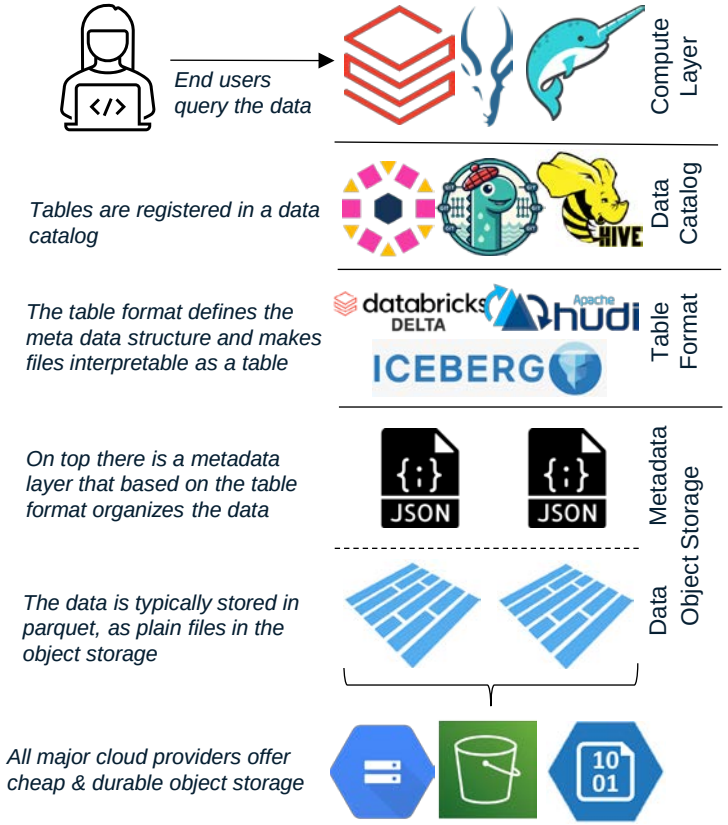
The current trend is towards datalake(houses) replacing traditional data warehouses systems

The latter are considered expansive and don’t scale well

Traditional data warehouses systems suffer from vendor lock-in



New table formats (e.g. Iceberg) solve all problems of old open table format Hive



Lessons learned

- Traditional data warehouse system suffer from many problems:
  - Vendor locking
  - Blackbox behaviour
  - “cool new thing” lockout
- Modern table formats are a standard to how to organize data files around your table.

Thank you for your attention

Please reach out in case of questions/suggestions

Email

jannic.cutura@dsti.institute

Phone

+49 170 906 6238

Website

www.janniccutura.net

Replication files

Hosted on GitHub