NYU Center for Data Science

Week 06.1: Column Oriented Storage

DS-GA 1004: Big Data

This week

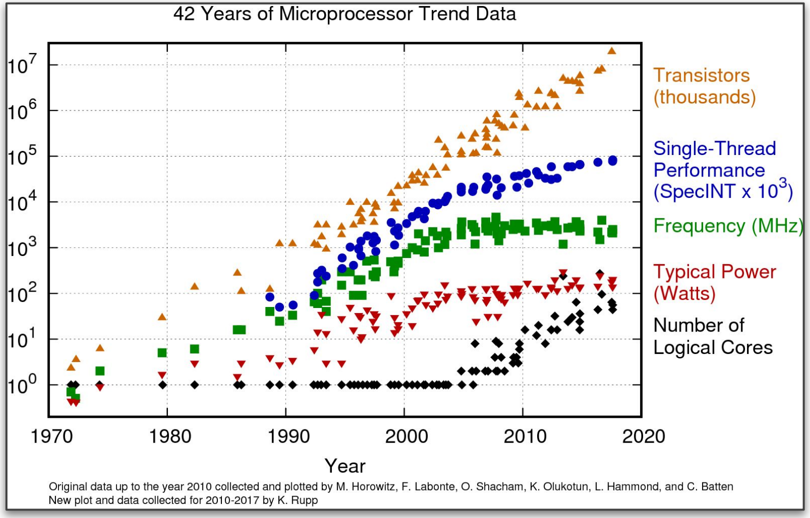
- **Column-oriented storage**
- Dremel and Parquet

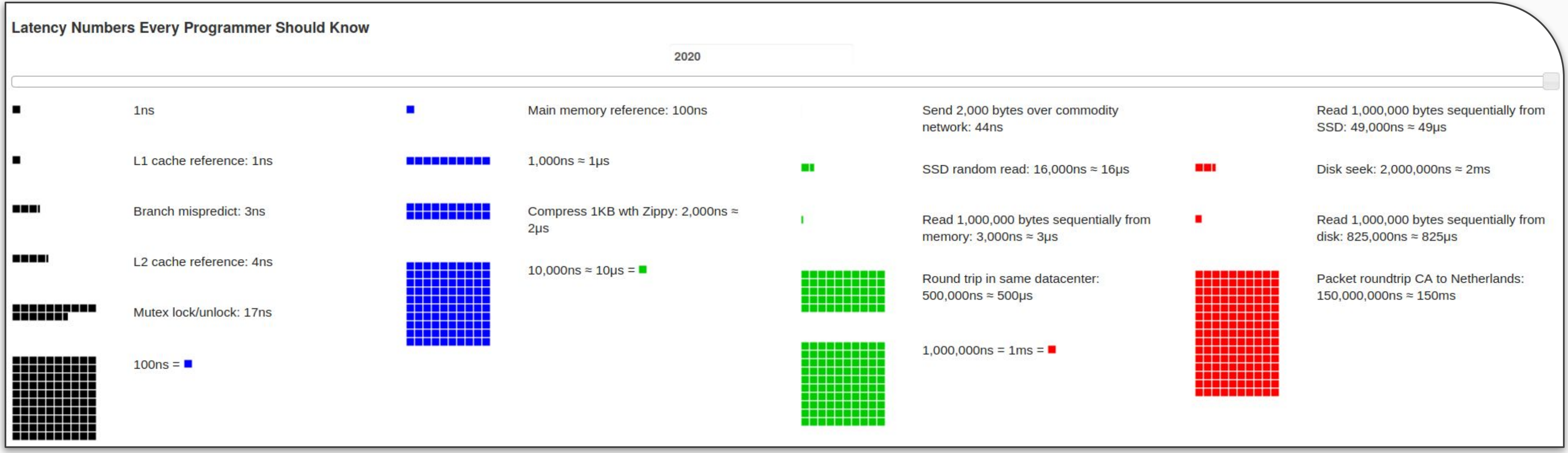
TLDR: parallelism isn't everything.

Data structures are still important!

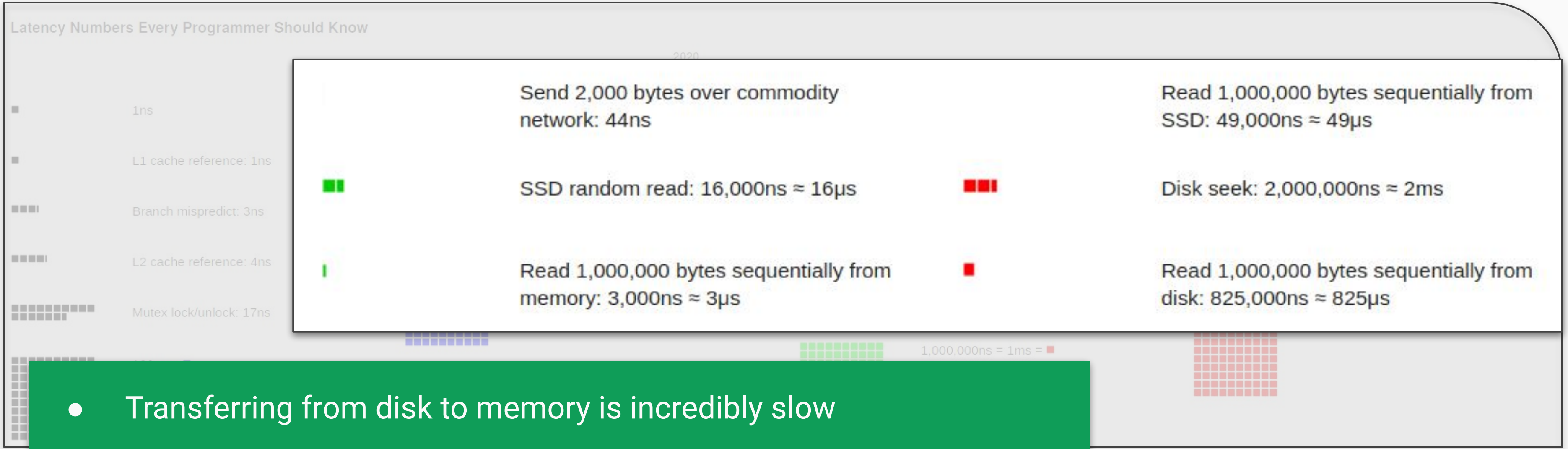
Column store history

- Idea goes back to the 1970s and 1980s
  - Transposed files [Batory, 1979]; Cantor [Karasalo & Svensson, 1983]
- Resurgence in the 2000s
  - MonetDB [Boncz & Kersten, 2002]
  - C-Store [Stonebraker et al., 2005]
  - VectorWise [Idreos et al., 2008]
- Why the resurgence?
  - Increased CPU speed + deep pipelining
  - Stagnant storage speed





https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html



- Transferring from disk to memory is incredibly slow
- Sequential memory reads are faster due to cache pre-fetching
- Strategies:
  - Transfer fewer bytes
  - Use predictable and contiguous memory access patterns

https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Row-oriented storage

- Relational data can be logically **grouped by rows**
  - Each record (tuple) represents a data point
  - Example: CSV files

- This is good if you want to process an entire record at a time

- Also good for appending data

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

Querying row stores

SELECT * **FROM** Dinosaur **WHERE** Awesome = True

for row **in** Dinosaur:
    if row.Awesome = True:
        emit row

- Each row is loaded from storage (disk)
- Attributes are inspected
- Rows that pass are sent down-stream

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

Indices can help, but...

```
SELECT Species FROM Dinosaur
WHERE Awesome = True
```

Now with index on Dinosaur.Awesome

```
for row in Dinosaur[Awesome = True]:
    emit row.Species
```

- An index can help locate rows
- But it still involves pulling an entire row, even if we only want one column
- Loading data from **disk is slow**!

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

Column-oriented storage

- Each column is stored on its own
- Values in a column have **constant type**
    - Disk access patterns become more regular
    - Improves cache locality
    - Enables compression and vectorized processing

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

Example

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

Row-oriented

id, Species, Era, Diet, Awesome

1, T. Rex, Cretaceous, Carnivore, True

2, Stegosaurus, Jurassic, Herbivore, True

3, Ankylosaurus, Cretaceous, Herbivore, True

Column-oriented

id: [1, 2, 3]

Species: [“T.Rex”, “Stegosaurus”, “Ankylosaurus”]

Era: [“Cretaceous”, “Jurassic”, “Cretaceous”]

Diet: [“Carnivore”, “Herbivore”, “Herbivore”]

Awesome: [True, True, False]

Compression

id	Species	Era	Diet	Awesome	Mass
1	T. Rex	Cretaceous	Carnivore	True	8000
2	Stegosaurus	Jurassic	Herbivore	True	4000
3	Ankylosaurus	Cretaceous	Herbivore	False	4000

- **Records** have **heterogeneous** types
- A **single column** only has **one type**
- **Low entropy** in a column ⇒ **compression**
  - Compressed columns take less space
  - Compressed columns are **cheaper to load**
  - Sometimes we can **compute directly on compressed columns**!
- **But what kind of compression should we use?**

Dictionary encoding

id	Species	Era	Diet	Awesome	Mass
1	T. Rex	Cretaceous	Carnivore	True	8000
2	Stegosaurus	Jurassic	Herbivore	True	4000
3	Ankylosaurus	Cretaceous	Herbivore	False	4000

- Useful when you have an attribute which takes **few distinct values**
- Replace **string values** by **string identifiers**
- Column now has **uniform data width**
  ⇒ better cache locality!
- String matching can be done on the **dictionary**, not each row

Cretaceous
Jurassic
Cretaceous
Cretaceous
Triassic
Cretaceous
…

0
1
0
0
2
0
…

Cretaceous ⇔ 0
Jurassic ⇔ 1
Triassic ⇔ 2

Bit-packing

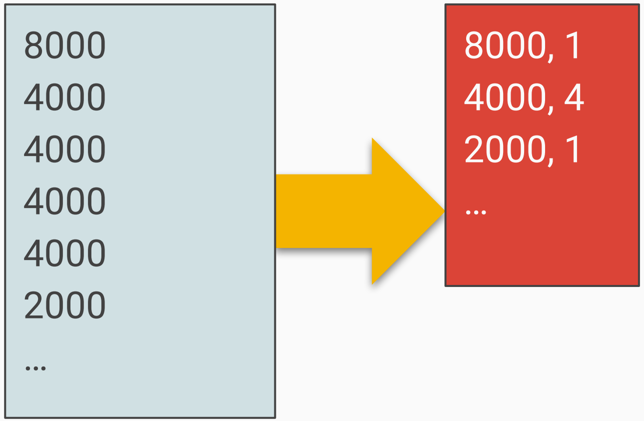
- Integers usually consume 4, or 8 bytes (32 or 64 bits)
- **Bit-packing** squeezes **small integers** together

Values	0	1	0	2	1	1
8-bit (binary)	0000 0000	0000 0001	0000 0000	0000 0010	0000 0001	0000 0001
Compressed	0001 0010	0101				

- Matching and comparing can be done on **compressed values**

Run-length encoding

id	Species	Era	Diet	Awesome	Mass
1	T. Rex	Cretaceous	Carnivore	True	8000
2	Stegosaurus	Jurassic	Herbivore	True	4000
3	Ankylosaurus	Cretaceous	Herbivore	False	4000



- Useful when you have long runs of a constant value
- Convert **sequence of values** to tuples **(value, # repetitions)**
- Sums, averages, counts, etc can all be done on **compressed values**

Compression schemes abound...

- Frame of reference coding
  - 1004, 1005, 1006 ⇒ **1000** | 4, 5, 6
- Delta coding
  - 1004, 1005, 1006 ⇒ **1004** | +0, +1, +1
- Lempel-Ziv-Welch (LZW) compression

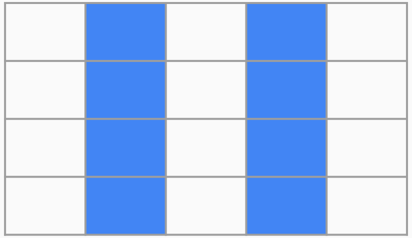
Compression schemes can be **combined**!
Delta + bit packing
Dictionary + Run-length encoding

Main trade-off is ***space efficiency*** vs. ***complexity of querying/processing***.

Column storage take-aways

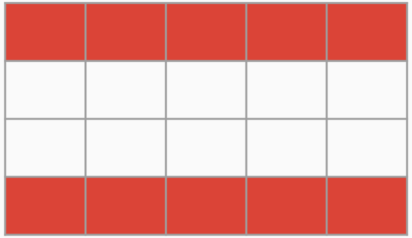
Pros:

- Can be much faster when you only want a **subset of attributes**
- Higher **storage efficiency** and **throughput**
- Collecting **data of the same type** enables compression and better access patterns



Cons:

- Reconstructing full tuples can be slow
  - Not great for **record-oriented** jobs
- Writes / deletion can be slow
- **Handling non-tabular data is tricky**



What if our data isn’t tabular?

Come back for part 2…