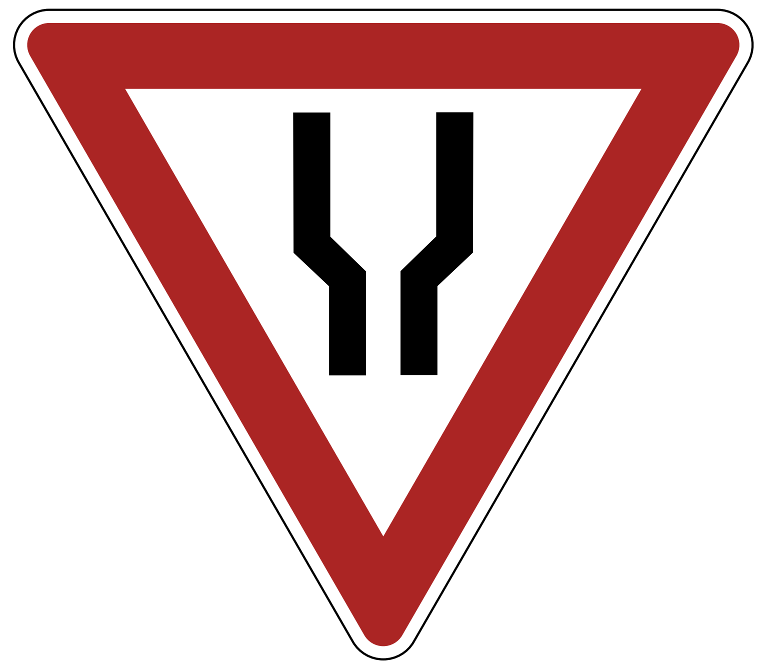
Programming with Data Bootcamp: Lecture 7

http://dsg.csail.mit.edu/6.S079/



Slides courtesy of Sam Madden / Tim Kraska (6.S079)

Key ideas:
Performance Bottlenecks
Data Layouts / Data Locality

Overview

- High level tools like Python are fine for many problems but may be too slow, especially as you scale up problem size
- Typically requires optimization and redesign
- Some strategies
  - Buy more hardware
  - Use a different runtime
  - Improve implementation
- Today we will focus on some simple data-oriented improvements; parallelism and algorithmic tricks in later lectures

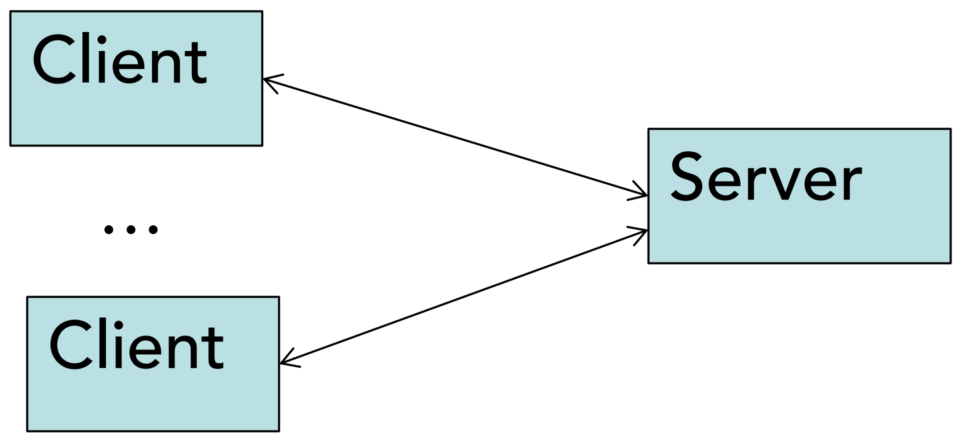
General Approach

- Find the bottleneck
  - Most programs have several stages
  - Some may be I/O based, some CPU based
- Improve performance of bottleneck
- Iterate
  - Did the bottleneck change?

How Slow is Slow?

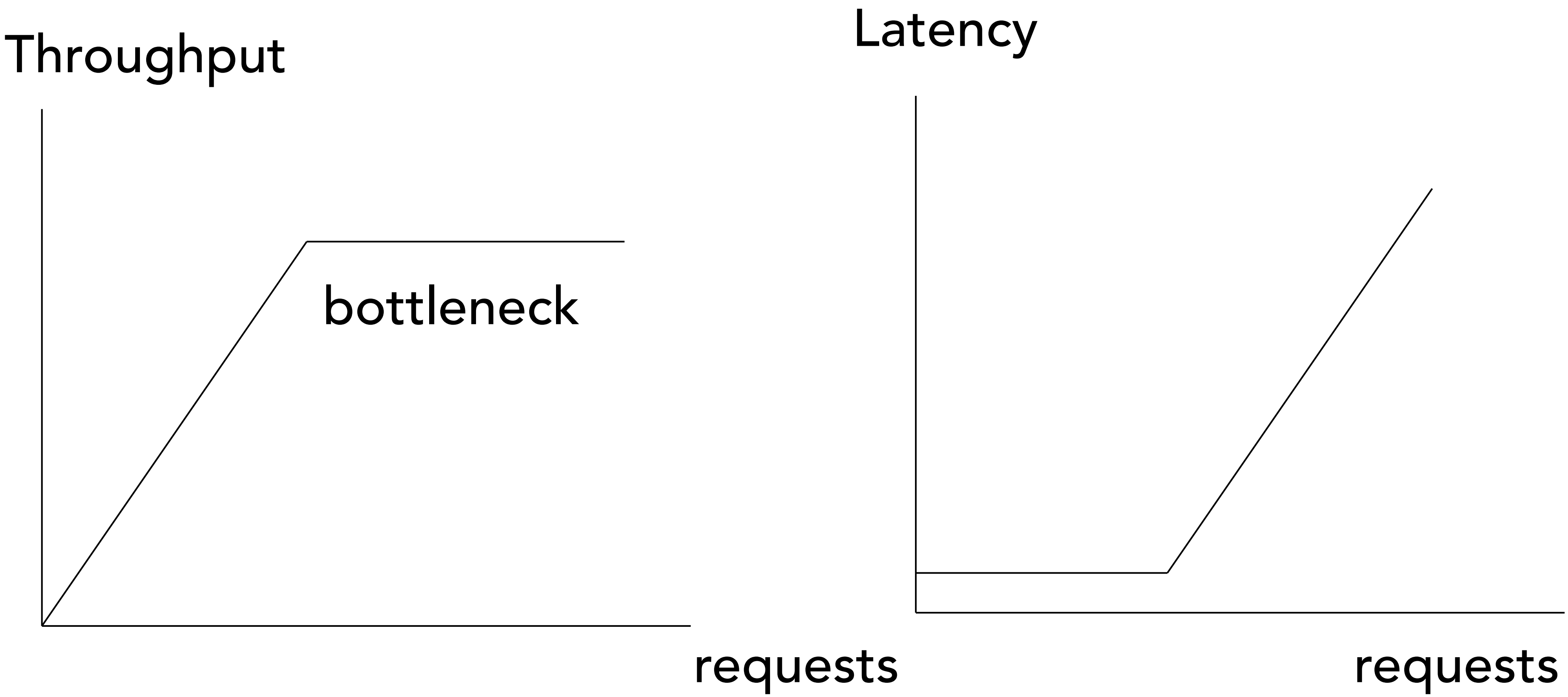
- Different applications have different performance demands
- In an online setting, e.g., serving a web page, 100ms may be too long
- For an interactive dashboard, 1s may be too long
- For an ML prediction, minutes may be too long

Performance metrics



- Performance metrics:
  - **Throughput**: request/time for many requests
  - **Latency**: time / request for single request
- Latency = 1/throughput?
  - Often not; e.g., server may have two CPUs

Heavily-loaded systems



- Once system busy, requests queue up

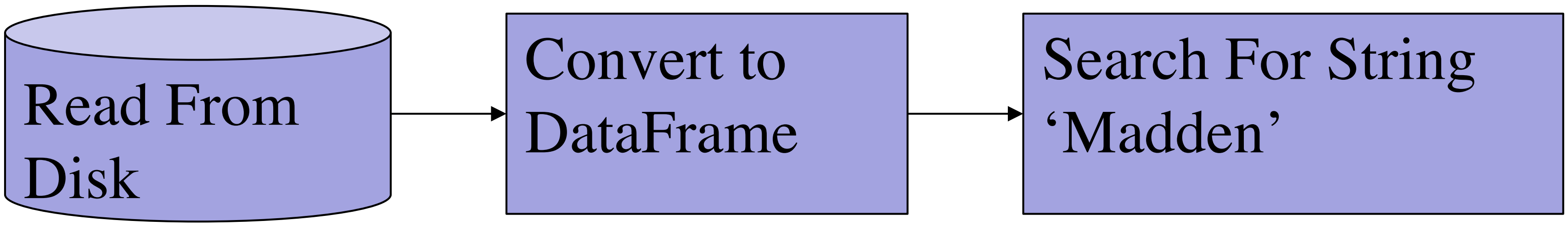
Approaches to finding bottleneck

300 MB file

```
df = pd.read_csv(PATH, delimiter='|',
      header=None, names=header)
print df[df['NAME'].str.contains("MADDEN")]
```

- Measure utilization of each resource
  - CPU is 100% busy, disk is 20% busy
  - CPU is 50% busy, disk is 50% busy, alternating
- Model performance of your approach
  - What performance do you expect?
- Guess, check, and iterate
  - Don’t prematurely optimize

How Long Do We Expect This To Take?



- I/O vs CPU
- Which will dominate?

Some Tools

- print statements / timing
- top / system profilers
- code profilers

Python code profile

```
python3 -m cProfile -o my_program.prof slow_pandas.py
snakeviz my_program.prof
```

Why Is This So Slow?

- Takes 7+ seconds. Why?
- Seems to be ~6s to load data frame, ~1s to perform search
- For loading, is it I/O? How long should reading from disk take?

Model Your Code

- How long should I/O take?
- How long should data loading take?
- How long should search take?

Important numbers

- Latency:
  - 0.000001 ms: instruction time (1 ns)
  - 0.0001 ms: DRAM load (100 ns)
  - 0.1 ms: LAN network packets (100 usec)
  - 0.2 ms: SSD random I/O (variable)
  - 10 ms: random HDD I/O
  - 25 ms: Internet east -> west coast
- Throughput:
  - 10,000 MB/s: DRAM
  - 4,000 MB/s: sequential SSD
  - 1,000 Mbits/s: Gbit LAN (or ~100 MB/s)
  - 500 MB/s: sequential HDD, or random SSD
  - 1 MB/s: random disk I/O

Disk Primer

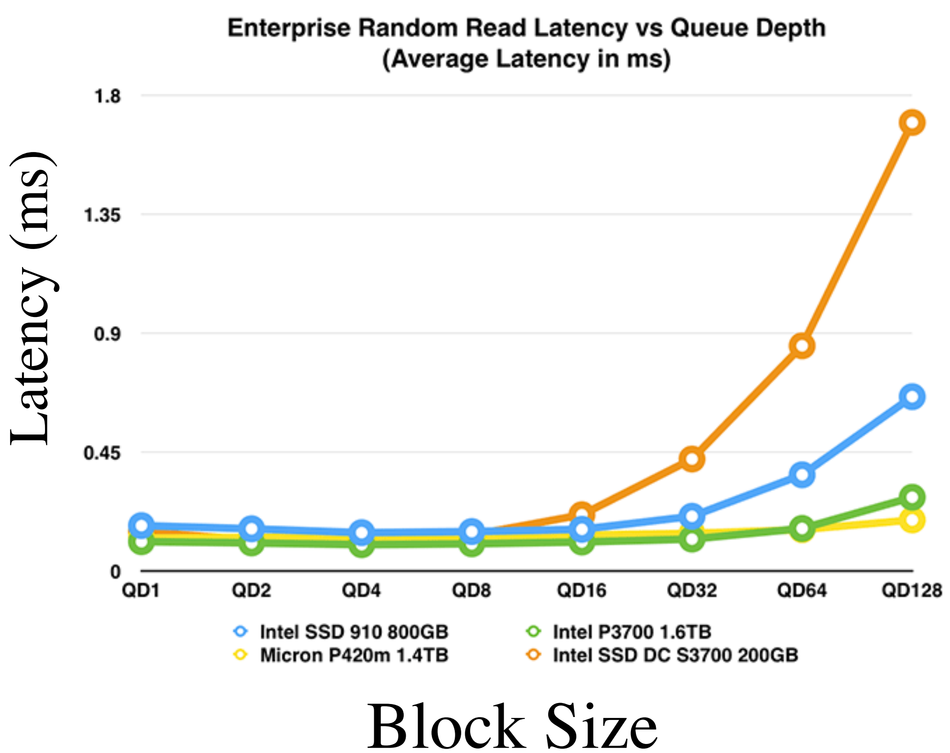
- Two main types of disks; hard disks(HDD) and solid state disks (SSD)
- Hard disks are rotating platters; cheaper and slower
- Both are block oriented, i.e., they allow reading or writing of blocks (usually a few KB)
- Unlike RAM, which is byte oriented

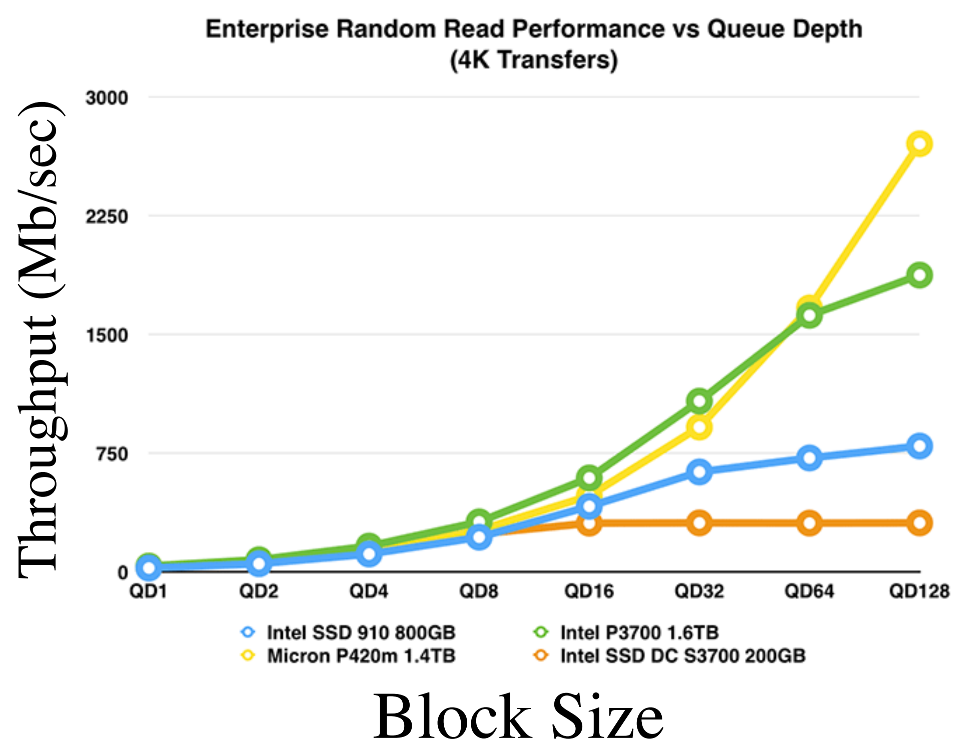
Solid State Disk (SSD)

- Faster storage technology than disk
  - Flash memory that exports disk interface
  - No moving parts
- Modern Apple 2TB SSD
  - Sequential read: 2.5 GB/sec
  - Sequential write: 250 MB/sec
  - Random 4KB read: 100K+/s (>400 GB/s)
    - See next slides
  - Random 4KB write: 10K+/s (>40 MB/s)

SSD Random Reads

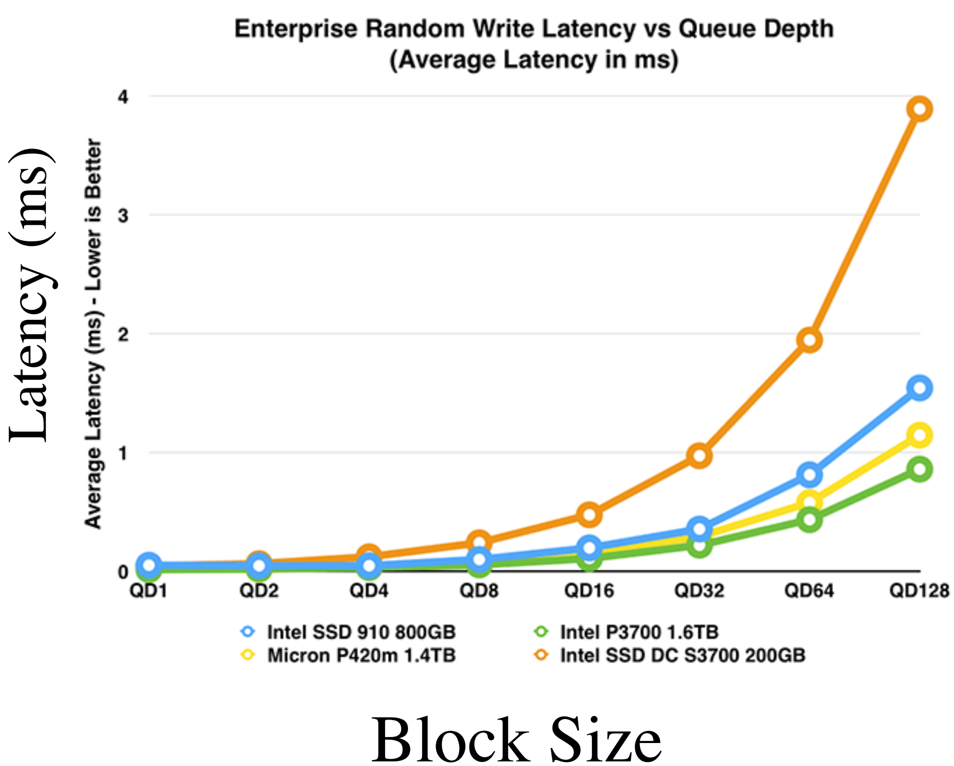
2014 Numbers

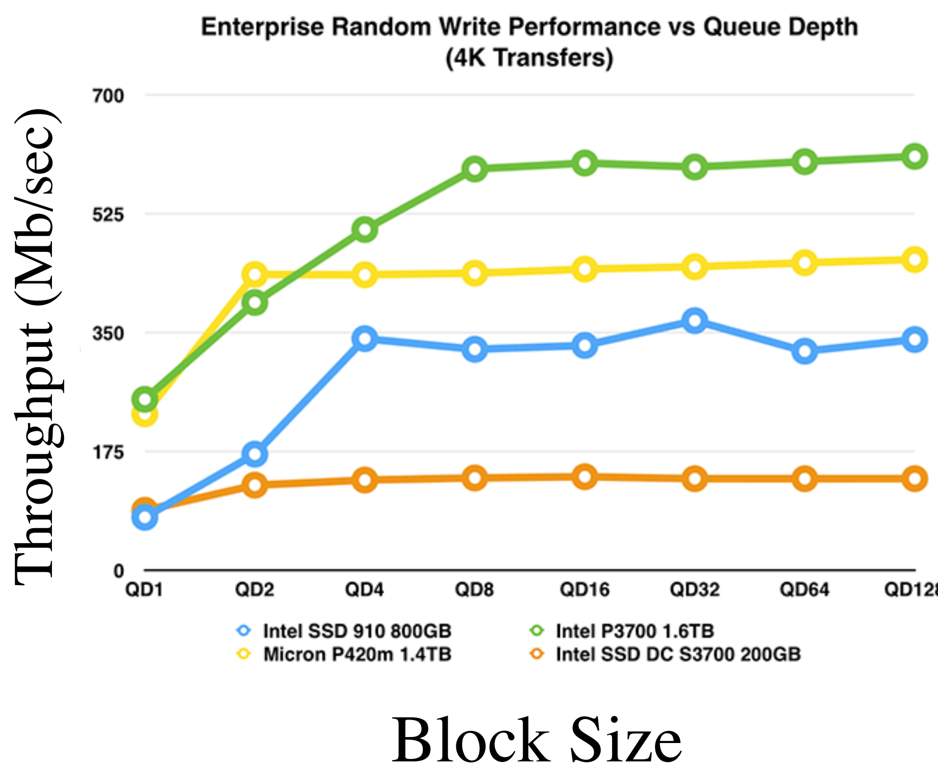




https://www.anandtech.com/show/8104/intel-ssd-dc-p3700-review-the-pcie-ssd-transition-begins-with-nvme/3

SSD Random Writes





SSDs and writes

- Write performance is slower:
  – Flash can erase only large units (e.g, 512 KB)
- Writing a small block:
  1. Read 512 KB
  2. Update 4KB of 512 KB
  3. Write 512 KB

- Controllers try to avoid this using aggressive caching, logging tricks

SSD versus HDD

- HDD: ~$100 for 4 TB
  - $0.025 per GB
- SSD: ~$200 for 2 TB
  - $1.00 per TB

HDD increasingly less common

- Many performance issues still the same:
  - Both SSD and Disks much slower than RAM
  - Avoid random small writes using batching

So How Much of 6s is I/O?

- Disk can read 1 GB/sec, 300 MB should take ~.3s. So disk I/O is not the issue!
  - But loading the data frame takes 6 s???
- What about CPU? 2M records, a few hundred instructions per record
  - ➔ ~400M instructions
  - Should take ~.2 seconds on a 2GHz proc
  - Actually takes 5-10x as long!

Fixing a bottleneck

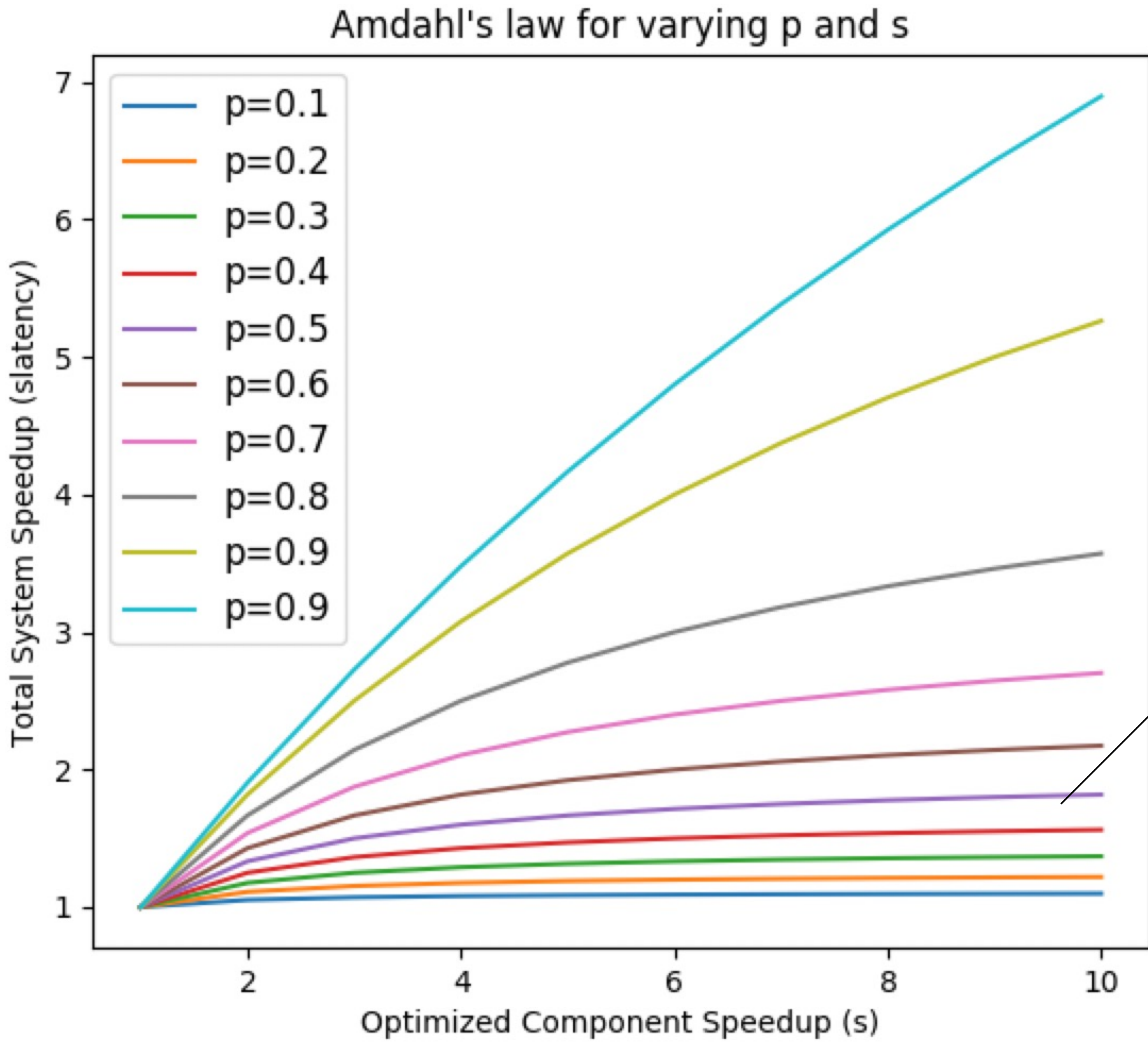
- Get better hardware
- Use better execution environment
- Find better algorithm
- Write better implementation; strategies
  - Indexing
  - Predicate push down
  - Early projection
  - Caching
  - Efficient joins
  - Partitioning & parallelism -- not today

What Improvement Can We Expect

- Always keep Amdahl's law in mind

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

$S_{latency}$ is the over all speedup in all stages of a task
s is the speedup on a stage of the task that we optimize
p is the original proportion of time the optimized stage took



If a component takes 50% of time, max speedup is 2x!

Clicker Question

Which do you think is going to result in best performance:

A. rewrite to use lower-level python instead of pandas, e.g., loops w/ readlines
B. rewrite in C
C. rewrite to use a relational database
D. none of these, pandas is best

Let's Try It

- Pandas version

read_time = 6.09, scan_time = 0.72

- Python loops

read_time = 11.72, scan_time = 0.71

- Rewrite in C

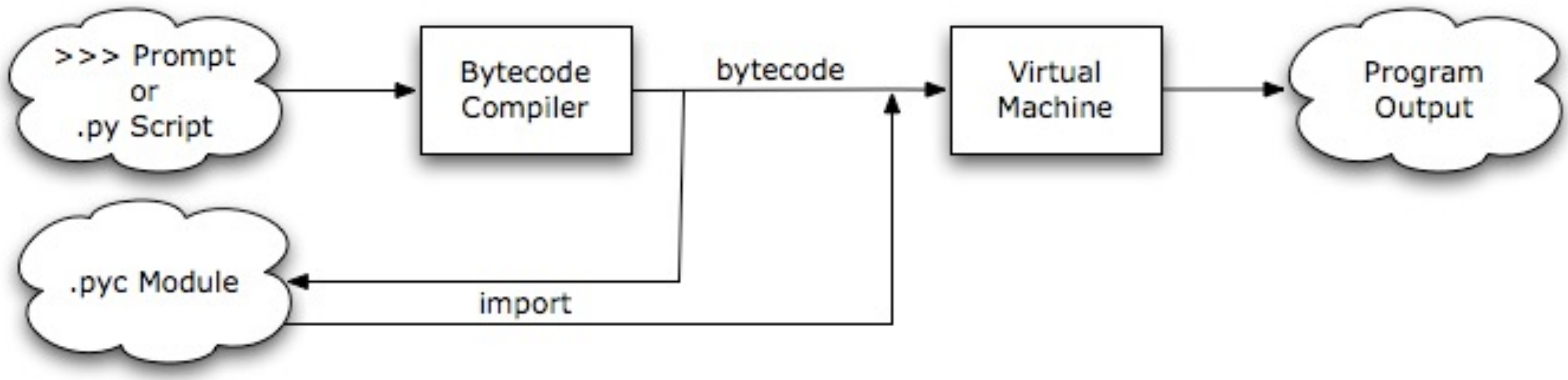
init_time = 0.00s, read_time = 1.58s, scan_time = 0.15s

- Use a Relational DB

```
donations=# \copy donations from
'indiv20/by_date/itcont_2020_20010425_20190425.txt' delimiter '|';
COPY 1976644
Time: 9345.116 ms (00:09.345)

donations=# select NAME, EMPLOYER, TRANSACTION_AMT from donations
where NAME ~ 'MADDEN' ;
Time: 405.118 ms
```

Why is Python So Slow



Virtual machine (VM) implementation is a loop that reads an instruction, and jumps to the code to execute the instruction

On modern CPUs this is very inefficient, because it results in many branch misses and poor processor cache locality

Python In Practice

- Loops python are very slow
  - Because it is an “interpreted” language, each operation takes 100’s of CPU cycles
  - Even though a CPU can run ~2B instructions per second, can only do about 5M loop iterations per second
- Pandas/numpy vectorized operations generally faster
  - Beware apply & co.

Summary

- Parsing data is the bottleneck
  - We will look at solutions next time
- Python is very slow
- Pandas is not bad
  - uses C implementations underneath
- Rewriting in C is painful, can be a big win
  - Can call into C from python if you have a specific algo you want to rewrite

Break

Algorithmic Bottlenecks

- Can we speed up text search?
- What about other kinds of slow algorithms?

Trigrams

1 23456

- MADDEN -> MAD, ADD, DDE, DEN …
- Index:

Sorted List

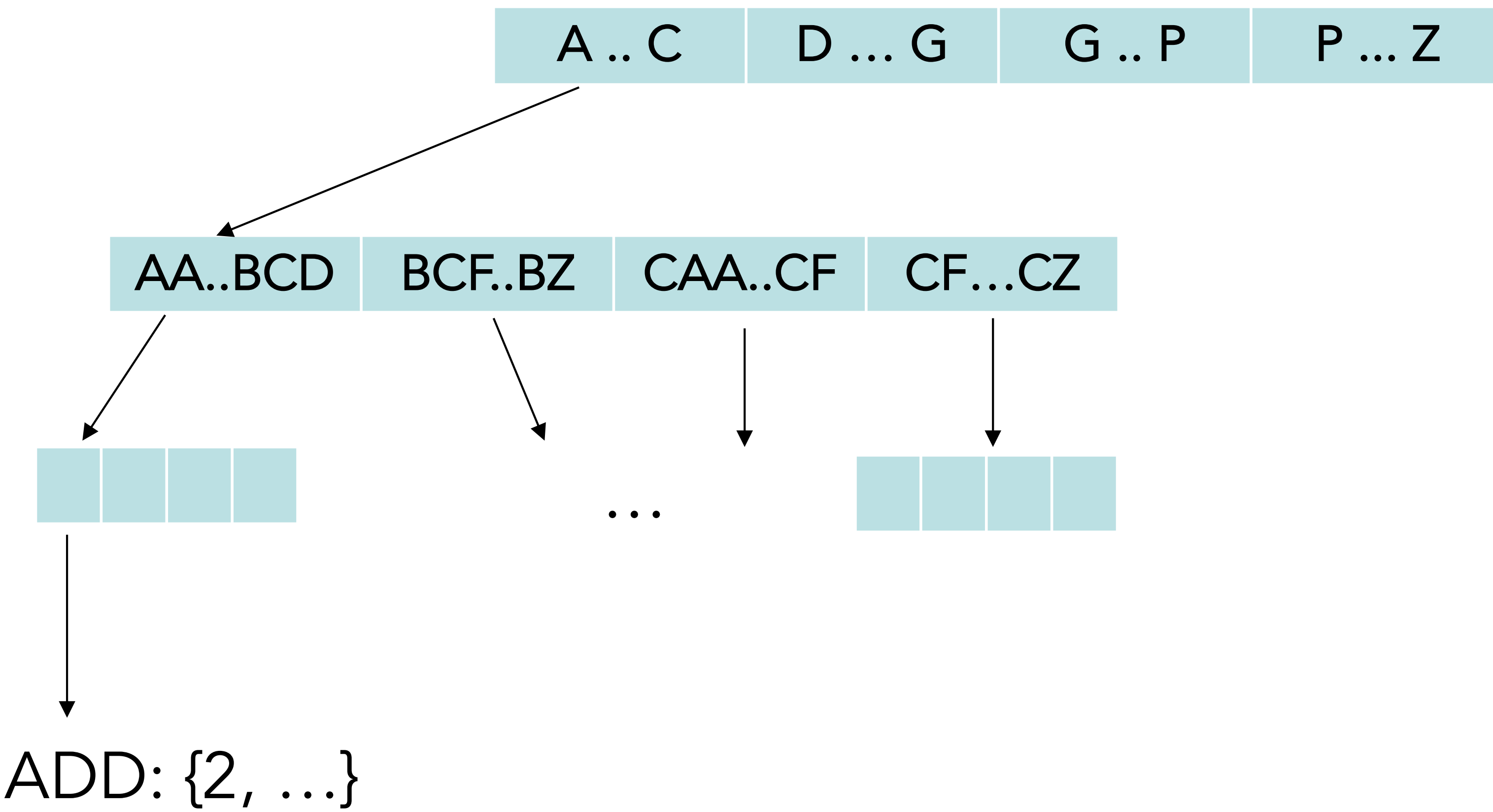
Trigram	Start Offsets in Text
ADD	2, …
DDE	3, …
DEN	4, …
MAD	1, …
…	

Lookup: MAD -> 1, DEN -> 4
These are consecutive, so found a match

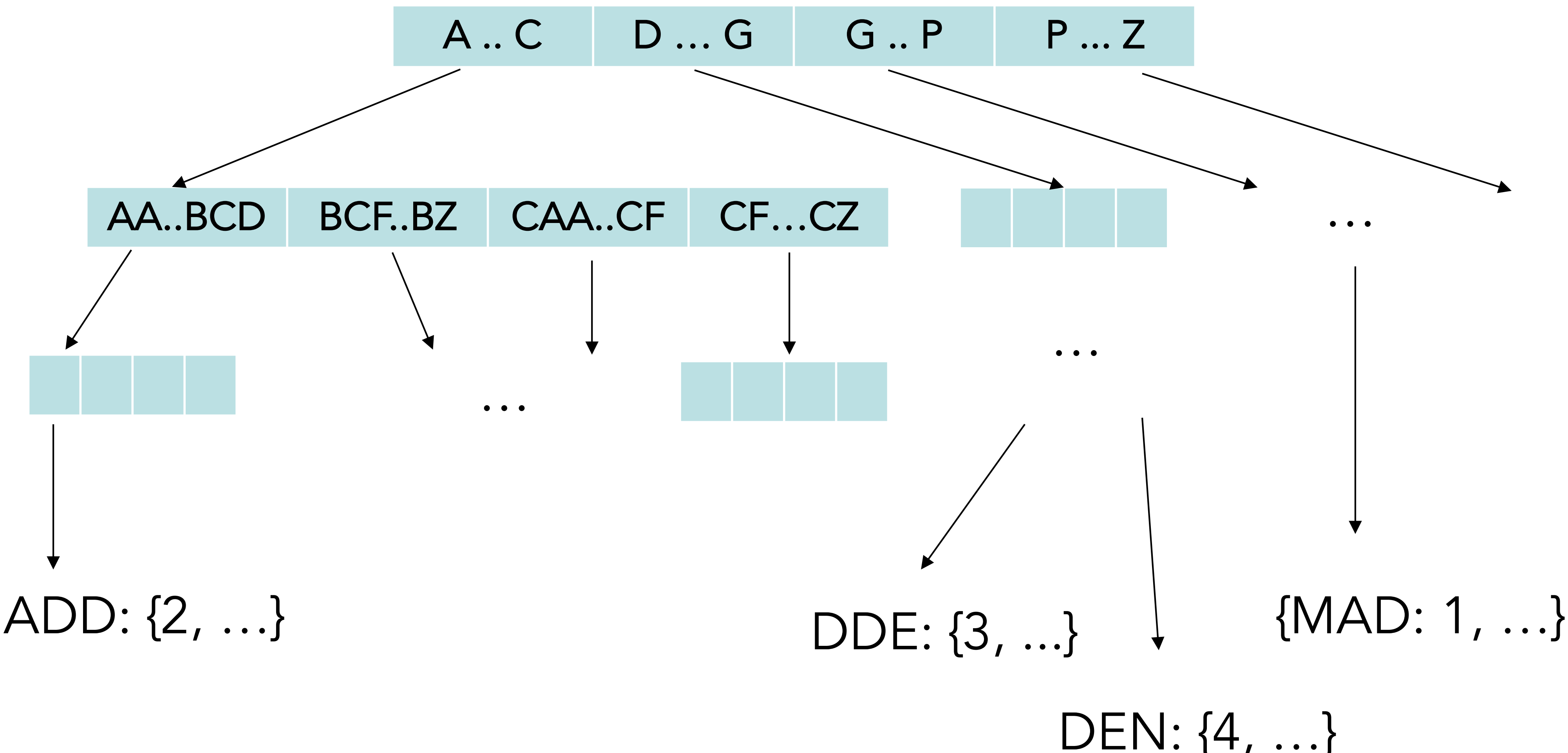
Tree Index

A .. C	D ... G	G .. P	P ... Z

Tree Index



Tree Index



What are advantages of tree organization over sorted list?

Creating Tree Index in Postgres

CREATE INDEX tbl_col_gin_trgm_idx ON donations USING gin (NAME gin_trgm_ops);

gin is a generic interface for describing tree indexes in Postgres

Performance

```
donations=# CREATE INDEX tbl_col_gin_trgm_idx  ON donations USING gin (NAME gin_trgm_ops);
```

Time: 8237.870 ms (00:08.238)

```
donations=# select NAME, EMPLOYER, TRANSACTION_AMT from donations where NAME ~ 'MADDEN' ;
```

Time: 2.129 ms

Other Common Algorithmic Bottlenecks

- What's wrong with this code?

```
start = time.time()

df = pd.read_csv(PATH, delimiter='|', header=None, names=header).loc[0:1000]
df2 = pd.read_csv(PATH2, delimiter='|', header=None, names=header).loc[0:1000]

end = time.time()
read_time = end-start

start = time.time()

matches = 0
for i,r in df.iterrows():
    for i2,r2 in df2.iterrows():
        if r.NAME == r2.NAME:
            matches = matches + 1

end = time.time()
join_time = end-start

print(f"got {matches} matches!")

print("read_time = %.2f, join_time = %.2f"%(read_time, join_time))
```

read_time = 11.13, join_time = 79.29

Solution 1

```
matches = 0
names = {}
for i,r in df.iterrows():
    if (r.name in names):
        names[r.name] = names[r.name] + [r]
    else:
        names[r.name] = [r]

for i2,r2 in df2.iterrows():
    if r2.NAME in names:
        matches = matches + len(names[r.name])
```

read_time = 11.19, join_time = 0.18

Solution 2

10x larger

```
df = pd.read_csv(PATH, delimiter='|', header=None, names=header).loc[0:10000]
df2 = pd.read_csv(PATH2, delimiter='|', header=None, names=header).loc[0:10000]
```

```
ans = []
ans = df.merge(df2, on="NAME")
```

read_time = 11.38, join_time = 0.07

Full 2M x 2M join

Activity Monitor
All Processes

CPU | Memory | Energy | Disk | Network

Process Name	Mem...	Threads	Ports	PID	User
Python	40.62 GB	8	21	84873	madden
WindowServer	4.48 GB	17	23,264	151	_windowserver
Mail	1.95 GB	9	7,555	470	madden
kernel_task	1.77 GB	309	0	0	root
Microsoft PowerPoint	1.59 GB	41	6,806	475	madden
iTerm2	1.52 GB	7	481	492	madden
https://www.boston.com	1.35 GB	20	310	29996	madden
Maps	1.29 GB	8	476	489	madden
Xcode	1.00 GB	25	1,904	71701	madden
Safari	942.8 MB	15	11,354	476	madden
Zulip Helper (GPU)	828.0 MB	10	249	1397	madden
Firefox	768.1 MB	81	9,660	32505	madden
https://www.boston.com	763.3 MB	17	296	83390	madden
Sublime Text	757.5 MB	20	654	440	madden
TextEdit	718.1 MB	6	782	503	madden
https://docs.google.com, ...	702.2 MB	7	281	44412	madden
Safari Networking	648.1 MB	11	9,627	550	madden
Microsoft Word	607.1 MB	16	1,071	78928	madden
Calendar	580.4 MB	4	3,299	9178	madden
Quip Web Content	569.0 MB	4	112	58653	madden
Dropbox	557.8 MB	156	1,451	80194	madden
Notes	548.8 MB	5	642	505	madden

MEMORY PRESSURE

Physical Memory:	64.00 GB
Memory Used:	47.10 GB
Cached Files:	15.51 GB
Swap Used:	11.25 GB

App Memory:	32.31 GB
Wired Memory:	7.14 GB
Compressed:	7.64 GB

read_time = 11.79, join_time = 200.26

Let’s Try it In SQL

1. Base performance
2. Change algo from Merge to Hash
3. Increase Parallelism
4. Partition Data

SQL Advantages

- Many different implementations
- Declarative Control
  - Algorithm
    - Sort merge vs Hash
  - Parallelism
- Memory conscious – able to spill to disk

Summary

- Python is often slow
- Identifying performance bottlenecks is an art
  - Figure out if you have an I/O or CPU problem
  - Estimate expected performance
  - Remember Amdahl's law!
- Rewriting in low level languages can help
- Using more efficient data accesses can help
- Next time: How to efficiently store & access data on disk

What is Data Locality?

- Data "near" to data you've already accessed can usually be read more quickly

- Why?
  - **Blocking**: data is often arranged in blocks, and read a block at a time
    - If you just read a record in a block B, if the next record is in B that will be fast

  - **Pre-fetching**: hardware often retrieves the next N data items after the data item you just read

Example

- SELECT name FROM donations WHERE name ~ 'MAD%'

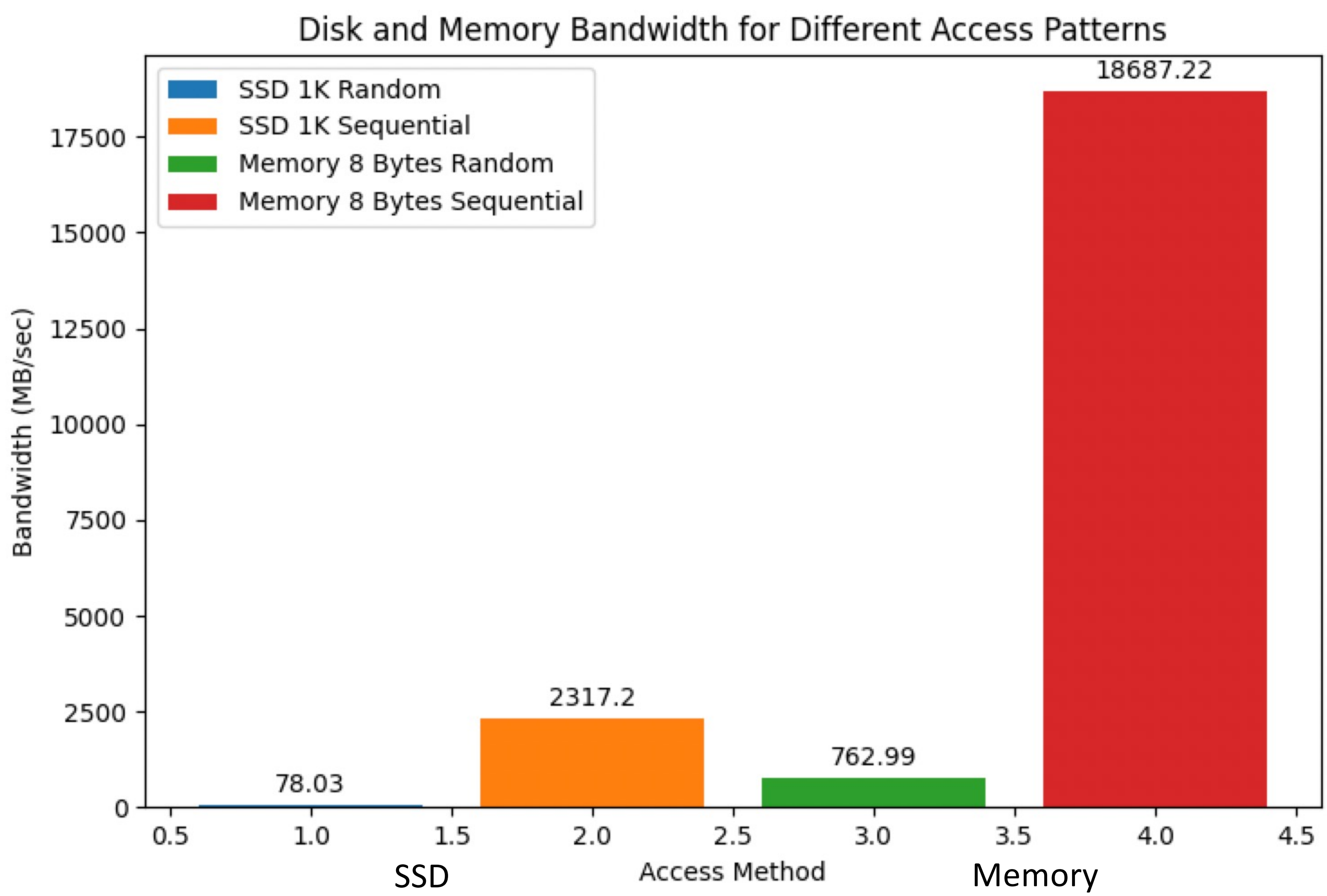
Sorted in name order
All "MAD" records on same few disk/memory blocks ➔ Sequential access to just those blocks

...
MACADAM
MADDAN
MADDEN
MADSEN
MADYAM
MARDEN
...

...
MADYAM
...
MADDEN
...
MARDEN
...
MADDAN
...
MACADAM
...
MADSEN
...

Not sorted
Each "MAD" records on different block
➔ Random access
(or sequential read through whole file)

Sequential Access is Much Faster



Is Data Transformation Worth the Price?

- Many of the techniques we will discuss only make sense if frequently re-accessing data
  - E.g., querying in a database
- Not worth spending a lot of time reorganizing data you're going to use once
  - E.g., to build an ML model
- But sometimes writing directly into a more efficient representation can benefit even infrequently read data

Data is N dimensional, Memory is Linear

- Have to “linearize” data somehow
- Examples:
  - Row-by-row
  - Column-by-column
  - Some more complicated N dimensional partitioning scheme
    - Quad-trees
    - Zorder

Linearizing a Table – Row store

C1	C2	C3	C4	C5	C6

Memory/Disk
(Linear Array)

R1 C1
R1 C2
R1 C3
R1 C4
R1 C5
R1 C6
R2 C1
R2 C2
R2 C3
R2 C4
R2 C5
R2 C6
R3 C1
R3 C2
R3 C3
R3 C4
R3 C5
R3 C6
R4 C1
R4 C2
R4 C3
R4 C4

Linearizing a Table –
Vertical Partitioning – aka ”Column Store”

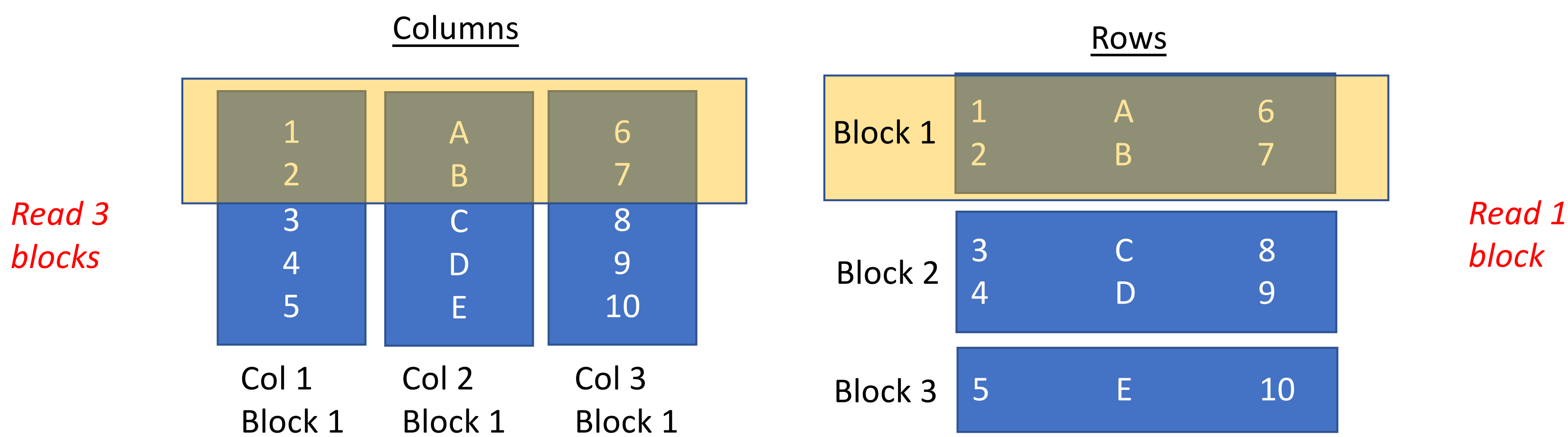
C1	C2	C3	C4	C5	C6

Memory/Disk
(Linear Array)

R1 C1
R2 C1
R3 C1
R4 C1
R5 C1
R6 C1
R1 C2
R2 C2
R3 C2
R4 C2
R5 C2
R6 C2
R1 C3
R2 C3
R3 C3
R4 C3
R5 C3
R6 C3
R1 C4
R2 C4
R3 C4
R4 C4

When Are Columns a Good Idea?

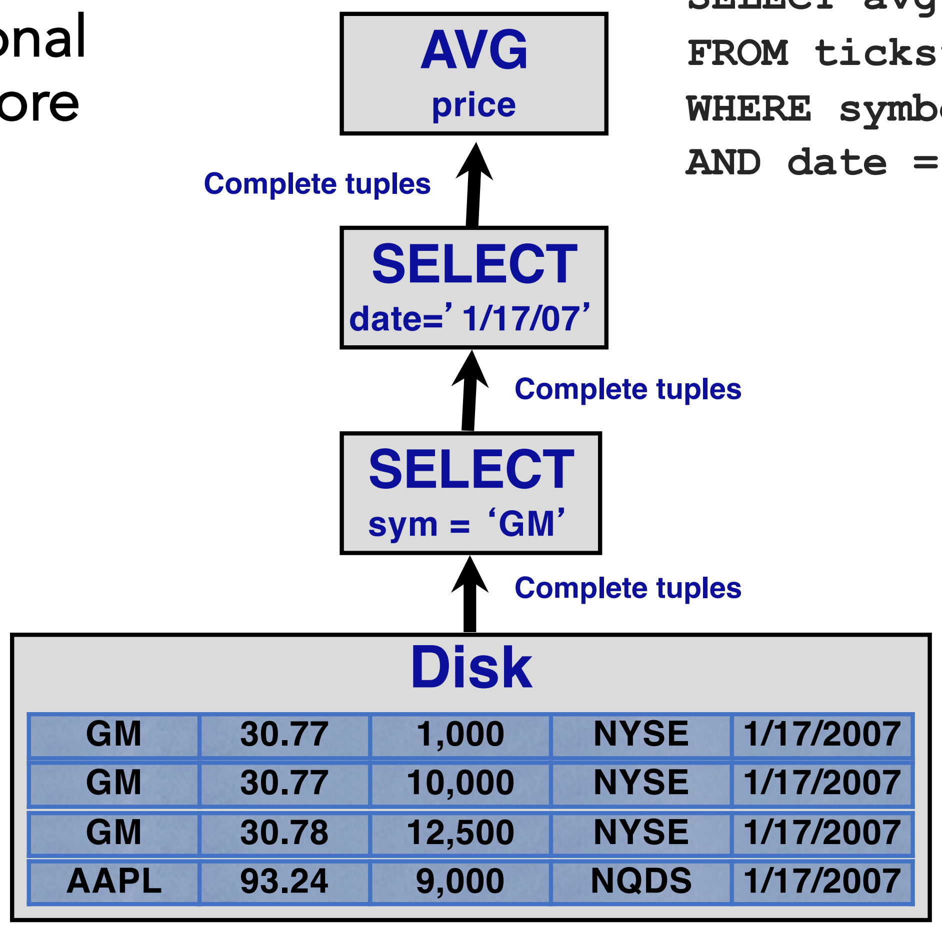
- When only a subset of columns need to be accessed
- When looking at many records
- Reading data from N columns of a few column-oriented records may be *worse* than using a row-oriented representation



Query Processing Example

- Traditional Row Store

```
SELECT avg(price)
FROM tickstore
WHERE symbol = 'GM'
AND date = '1/17/2007'
```

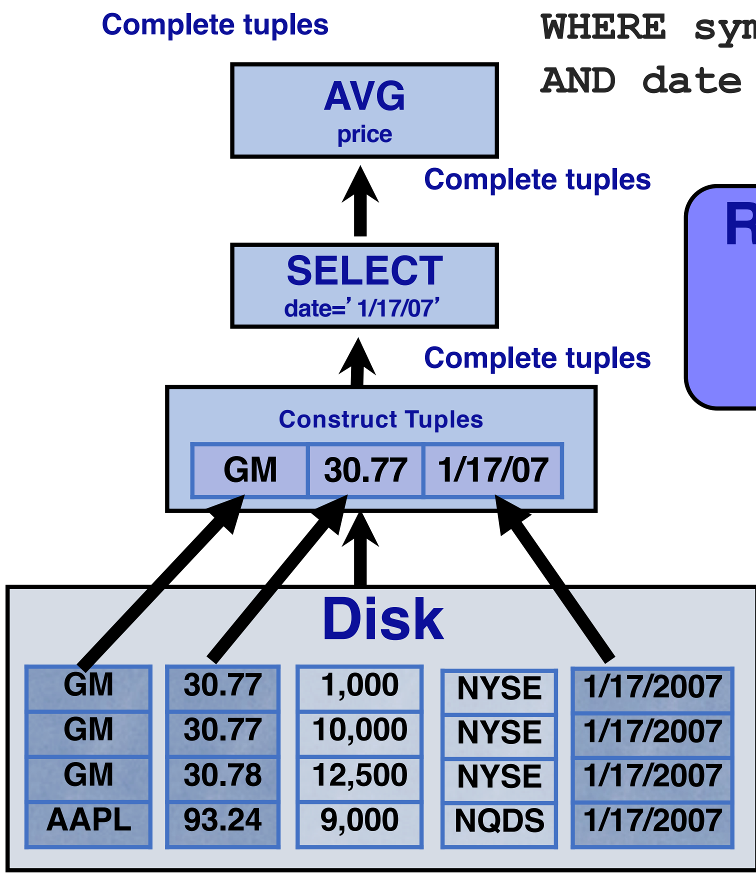


GM	30.77	1,000	NYSE	1/17/2007
GM	30.77	10,000	NYSE	1/17/2007
GM	30.78	12,500	NYSE	1/17/2007
AAPL	93.24	9,000	NQDS	1/17/2007

Query Processing Example

- Basic Column Store
- "Early Materialization"

```
SELECT avg(price)
FROM tickstore
WHERE symbol = 'GM'
AND date = '1/17/2007'
```

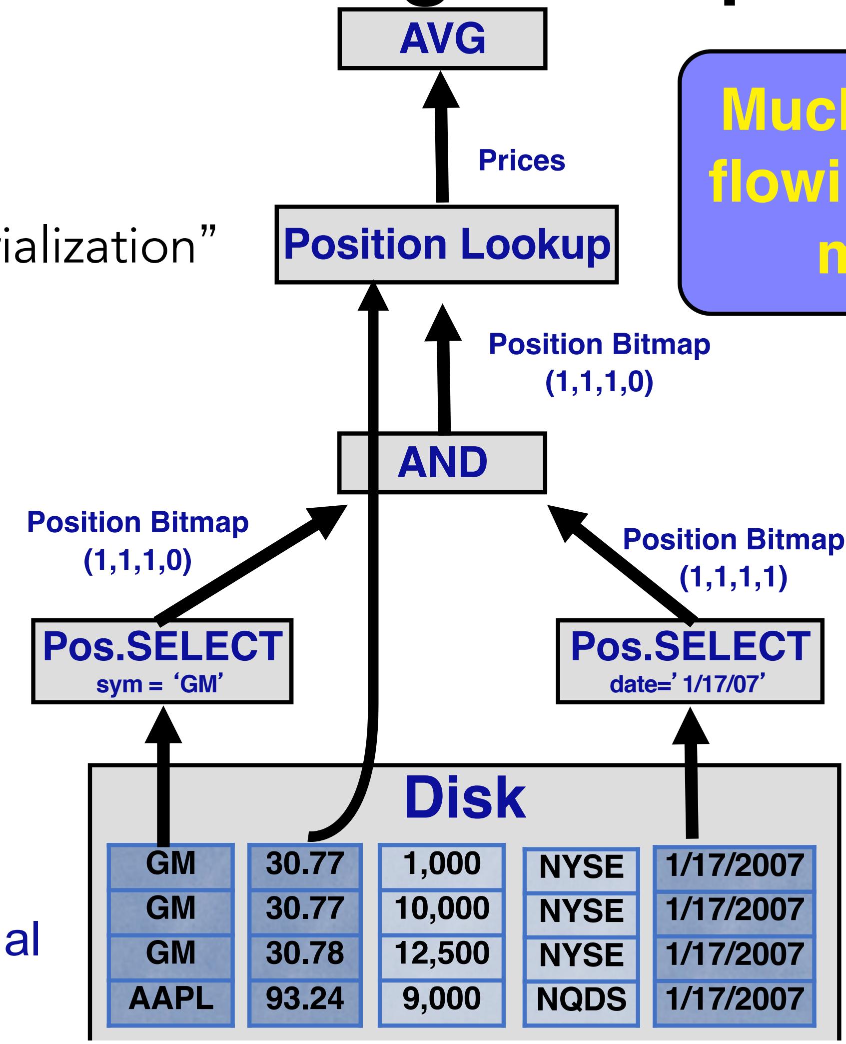


Row-oriented plan

Fields from same tuple at same index (position) in each column file

Query Processing Example

- C-Store
  - "Late Materialization"



Much less data flowing through memory

See Abadi et al ICDE 07

Parquet: Column Representation for Data Science

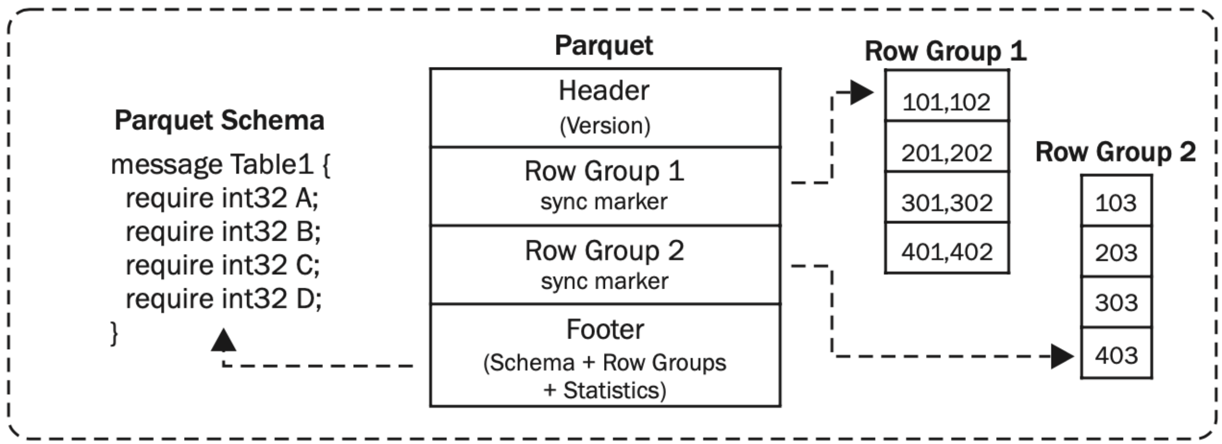
- Parquet is a column-oriented data form for storing tabular data

- Advantages are not just due to column orientation:
  - Data is stored in binary format, so more compact
  - Data is typed and types are stored, so parsing is much faster
  - Supports compression directly

Parquet Layout

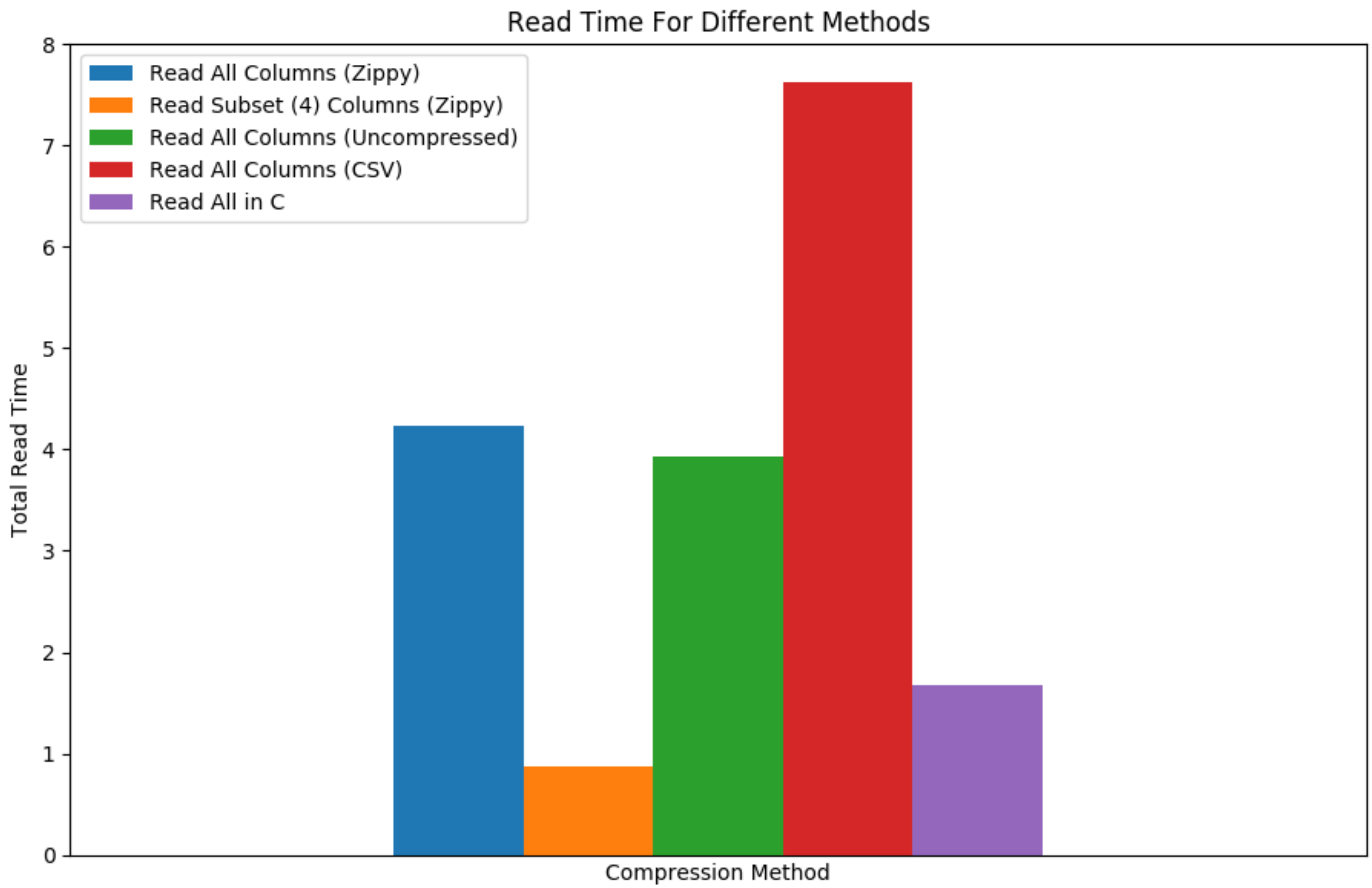
Table 1

A	B	C	D
101	201	301	401
102	202	302	402
103	203	303	403

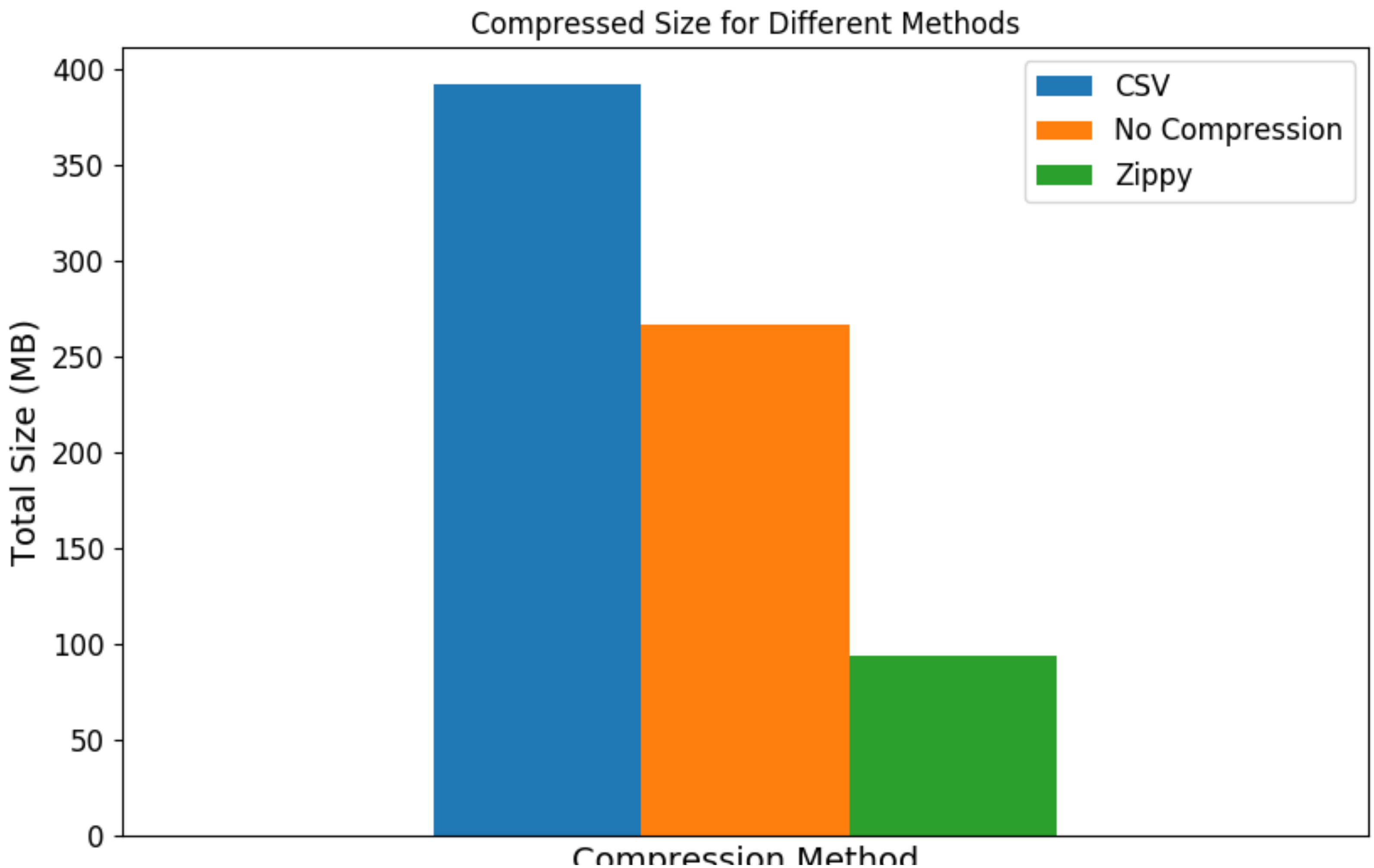


From “A Cost-based Storage Format Selector for Materialization in Big Data Frameworks”, Faisal et al

Parquet vs CSV Load Times



Parquet vs CSV File Sizes



Break

More Layout Tricks

- Data Partitioning
- Sorting
- Multi-dimensional Partitioning
- Compression
- Loading

Horizontal Partitioning

- Slice dataset according to some attribute

Date	Region	Profit
1/1/2019	NE	
1/2/2019	NE	
1/2/2019	SW	
1/2/2019	SE	
1/2/2019	NW	
1/3/2019	NE	
1/3/2019	SW	
1/3/2019	SE	
1/4/2019	SE	
1/4/2019	NW	
1/4/2019	NE	

Date	Region	Profit
1/1/2019	NE	

Date	Region	Profit
1/2/2019	NE	
1/2/2019	SW	
1/2/2019	SE	
1/2/2019	NW	

Date	Region	Profit
1/3/2019	NE	
1/3/2019	SW	
1/3/2019	SE	

Date	Region	Profit
1/4/2019	SE	
1/4/2019	NW	
1/4/2019	NE	

Postgres Example (From Lec 16)

```
 Partitioned table "public.donations_hash"
     Column      |        Type       | Collation | Nullable | Default | Storage  | Stats target | Description
-----------------+-------------------+-----------+----------+---------+----------+--------------+-------------
 cmte_id         | character varying |           |          |         | extended |              |
 amndt_ind       | character varying |           |          |         | extended |              |
 rpt_tp          | character varying |           |          |         | extended |              |
 transaction_pgi | character varying |           |          |         | extended |              |
 image_num       | character varying |           |          |         | extended |              |
 transaction_tp  | character varying |           |          |         | extended |              |
 entity_tp       | character varying |           |          |         | extended |              |
 name            | character varying |           |          |         | extended |              |
 city            | character varying |           |          |         | extended |              |
 state           | character varying |           |          |         | extended |              |
 zip_code        | character varying |           |          |         | extended |              |
 employer        | character varying |           |          |         | extended |              |
 occupation      | character varying |           |          |         | extended |              |
 transaction_dt  | character varying |           |          |         | extended |              |
 transaction_amt | character varying |           |          |         | extended |              |
 other_id        | character varying |           |          |         | extended |              |
 tran_id         | character varying |           |          |         | extended |              |
 file_num        | character varying |           |          |         | extended |              |
 memo_cd         | character varying |           |          |         | extended |              |
 memo_text       | character varying |           |          |         | extended |              |
 sub_id          | character varying |           |          |         | extended |              |
Partition key: HASH (name)
Partitions: donations_hash_1 FOR VALUES WITH (modulus 4, remainder 0),
            donations_hash_2 FOR VALUES WITH (modulus 4, remainder 1),
            donations_hash_3 FOR VALUES WITH (modulus 4, remainder 2),
            donations_hash_4 FOR VALUES WITH (modulus 4, remainder 3)
```

Sorting

- Can also order data according to some attribute

Date	Region	Profit
1/1/2019	NE	
1/2/2019	NE	
1/2/2019	SW	
1/2/2019	SE	
1/2/2019	NW	
1/3/2019	NE	
1/3/2019	SW	
1/3/2019	SE	
1/4/2019	SE	
1/4/2019	NW	
1/4/2019	NE	

Date	Region	Profit
1/1/19	NE	
1/2/19	NE	
1/3/19	NE	
1/4/19	NE	
1/2/19	NW	
1/4/19	NW	
1/2/19	SE	
1/3/19	SE	
1/4/19	SE	
1/2/19	SW	
1/3/19	SW	

Can both sort & partition

- E.g., partition on date, sort by region in each partition
  - Or vice versa
- Best choice depends on how we plan to access data, and on how much scanning we can avoid
  - If new data is arriving in some order (e.g., time) easy to write partitions in that order

Date	Region	Profit
1/1/2019	NE	

Date	Region	Profit
1/2/2019	NE	
1/2/2019	NW	
1/2/2019	SE	
1/2/2019	SW	

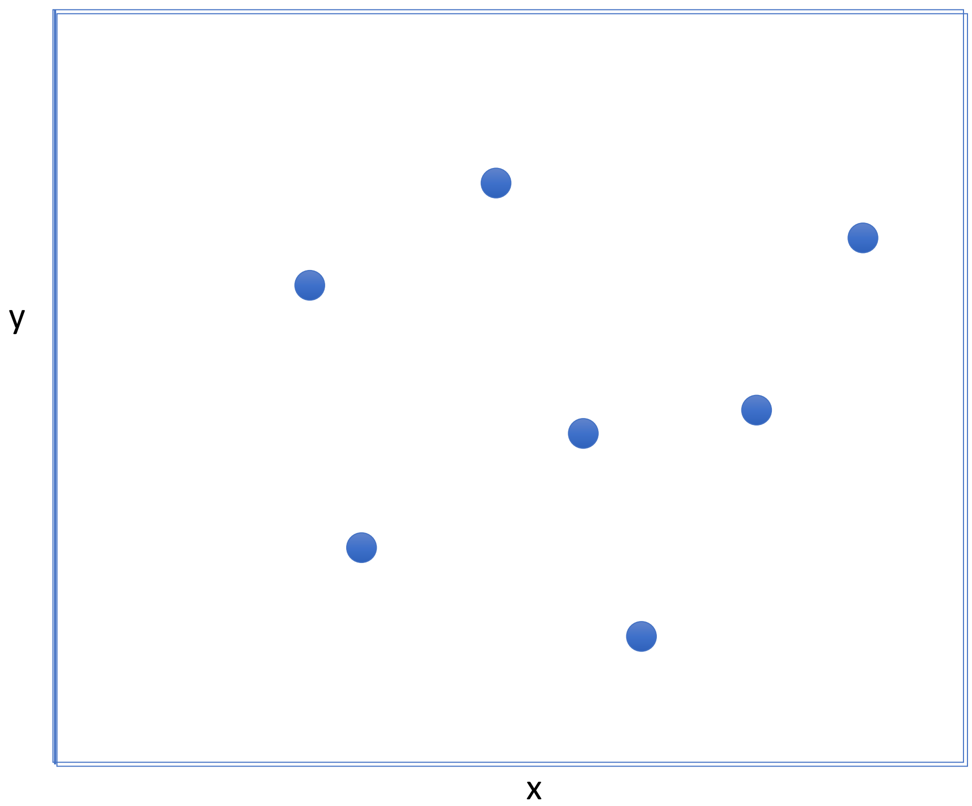
Date	Region	Profit
1/3/2019	NE	
1/3/2019	SE	
1/3/2019	SW	

Date	Region	Profit
1/4/2019	NE	
1/4/2019	NW	
1/4/2019	SW	

What if I want to partition on several attributes?

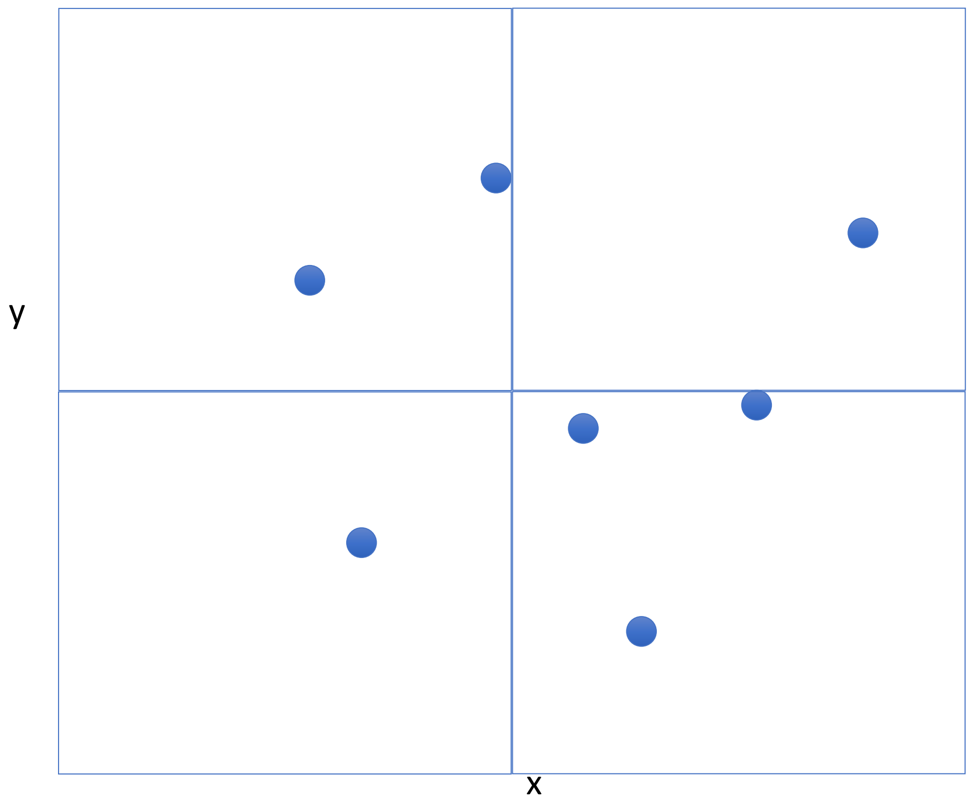
- Basic idea: ”tile” data into N dimesions
- 2 approaches:
- **Quad-tree:** recursively subdivide until tiles are under a target size
- **Z-order**: interleave multiple dimensions, order by interleaving

Quad-Tree



Quad-Tree

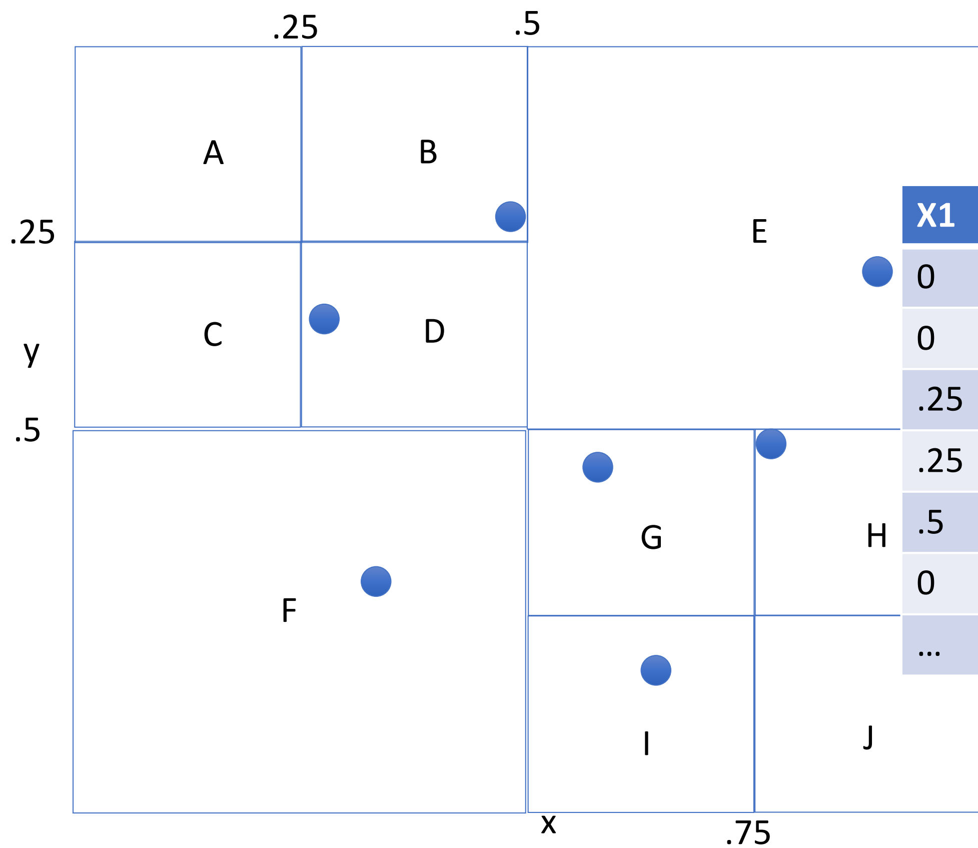
Recursively subdivide



Quad-Tree

Until partitions are of some maximum size

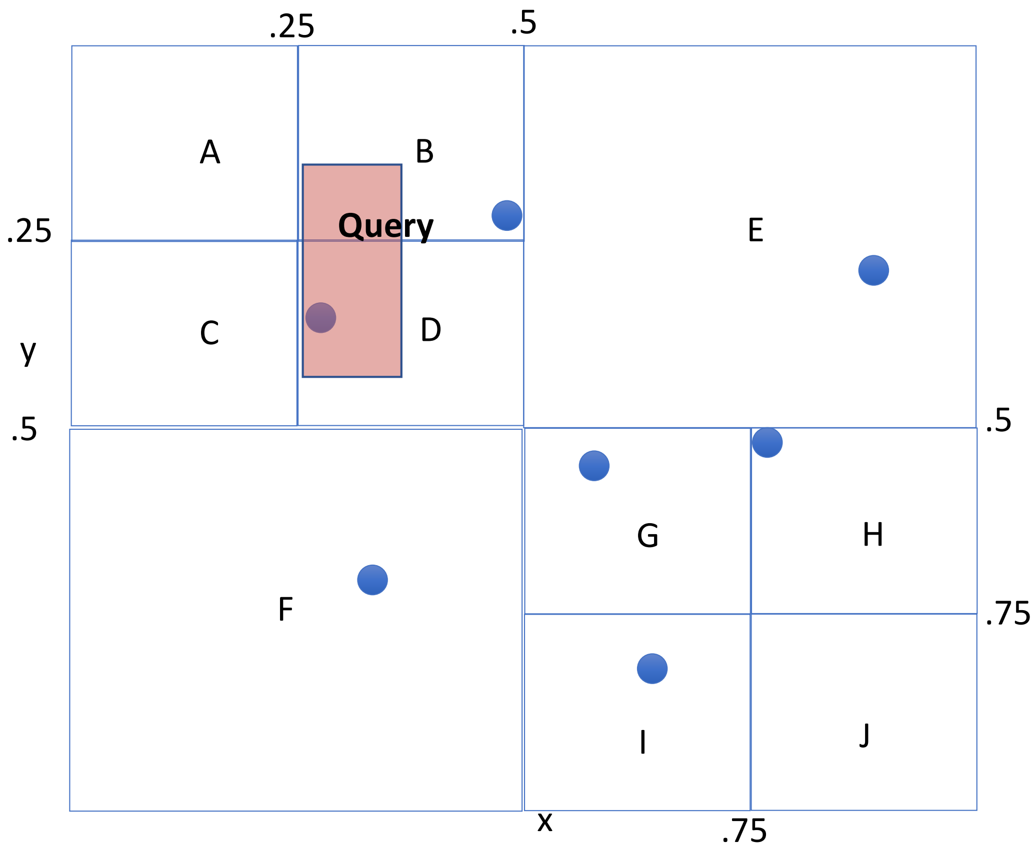
Index stores boundaries of rectangles, and pointers on disk



X1	X2	Y1	Y2	Part
0	.25	0	.25	A
0	.25	.25	.5	C
.25	.5	0	.25	B
.25	.25	.25	.5	D
.5	1	0	.5	E
0	.5	.5	1	F
…				

Quad-Tree

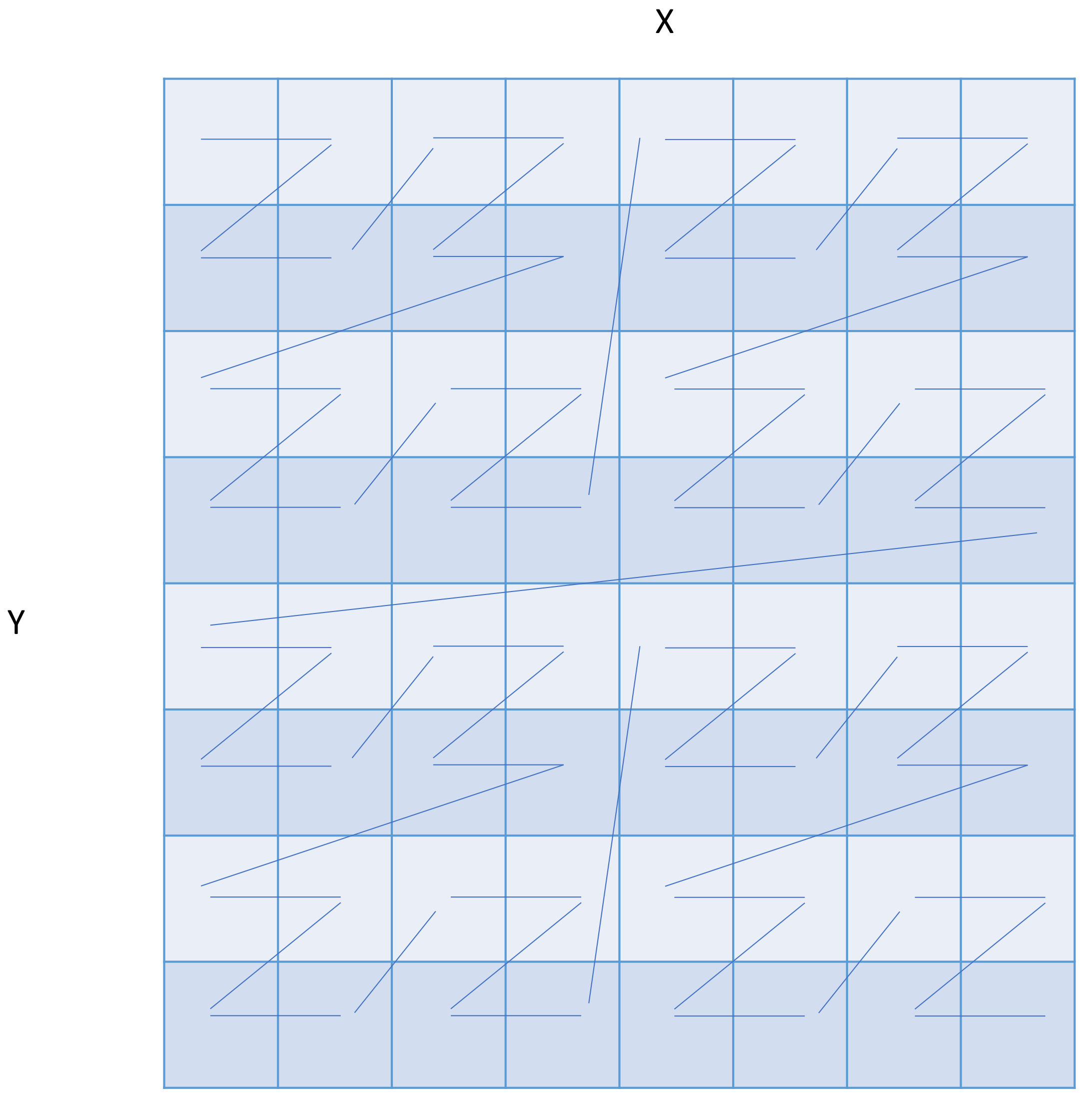
Until partitions are of some maximum size



Index stores boundaries of rectangles, and pointers on disk

X1	X2	Y1	Y2	Part
0	.25	0	.25	A
0	.25	.25	.5	C
.25	.5	0	.25	B
.25	.25	.25	.5	D
.5	1	0	.5	E
0	.5	.5	1	F
…				

ZOrder



Zorder Implementation

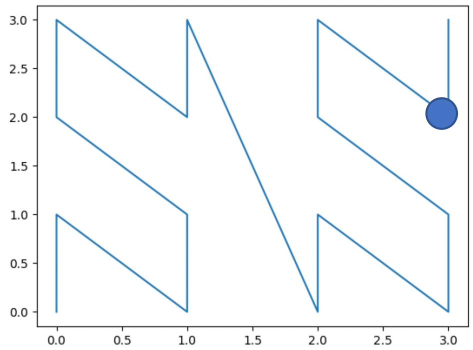
- To generate a Zorder, interleave bits of numbers

e.g., Zorder(3,2)

3 = 0011

2 = 0010

➔ 00001110 = 14

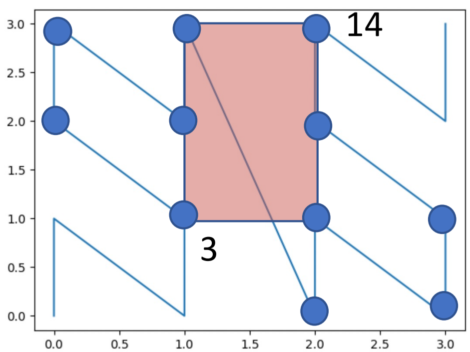


i	j	zorder	bits
0	0	0	[0, 0, 0, 0, 0, 0]
0	1	1	[0, 0, 0, 0, 0, 1]
1	0	2	[0, 0, 0, 0, 1, 0]
1	1	3	[0, 0, 0, 0, 1, 1]
0	2	4	[0, 0, 0, 1, 0, 0]
0	3	5	[0, 0, 0, 1, 0, 1]
1	2	6	[0, 0, 0, 1, 1, 0]
1	3	7	[0, 0, 0, 1, 1, 1]
2	0	8	[0, 0, 1, 0, 0, 0]
2	1	9	[0, 0, 1, 0, 0, 1]
3	0	10	[0, 0, 1, 0, 1, 0]
3	1	11	[0, 0, 1, 0, 1, 1]
2	2	12	[0, 0, 1, 1, 0, 0]
2	3	13	[0, 0, 1, 1, 0, 1]
3	2	14	[0, 0, 1, 1, 1, 0]
3	3	15	[0, 0, 1, 1, 1, 1]

Zorder Querying

- Support we want to look up data in Rectange((1,1),(2,3))

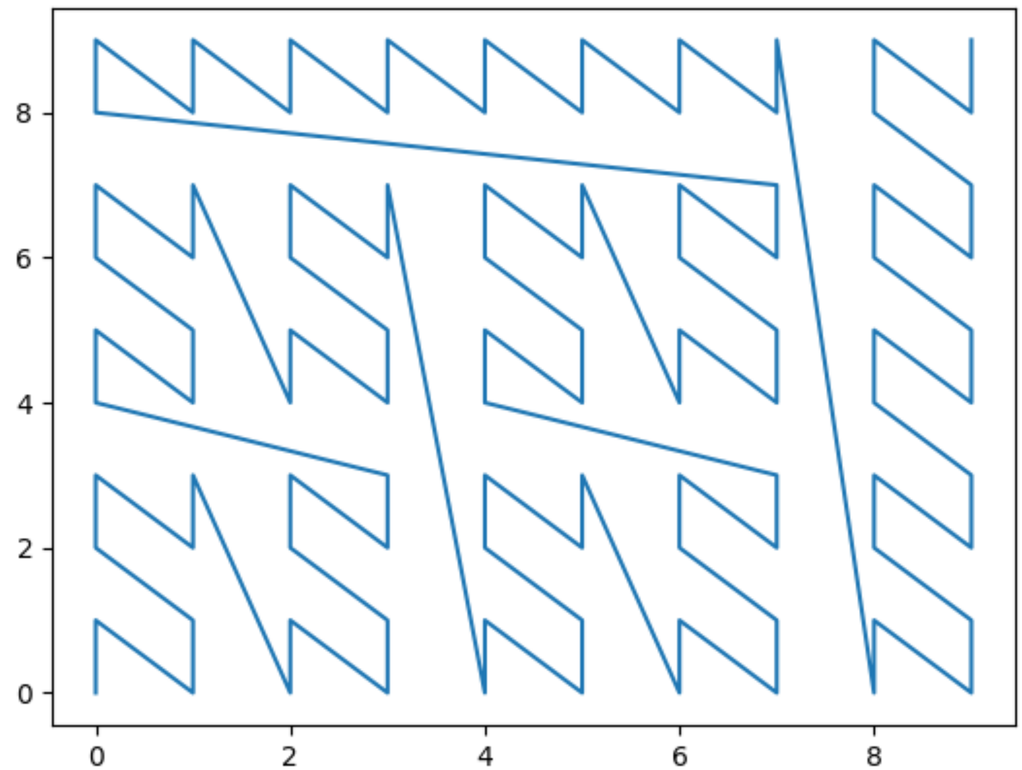
Zorder(1,1) = 0011 = 3
Zorder(2,3) = 1101 = 13



i	j	zorder	bits
0	0	0	[0, 0, 0, 0, 0, 0]
0	1	1	[0, 0, 0, 0, 0, 1]
1	0	2	[0, 0, 0, 0, 1, 0]
1	1	3	[0, 0, 0, 0, 1, 1]
0	2	4	[0, 0, 0, 1, 0, 0]
0	3	5	[0, 0, 0, 1, 0, 1]
1	2	6	[0, 0, 0, 1, 1, 0]
1	3	7	[0, 0, 0, 1, 1, 1]
2	0	8	[0, 0, 1, 0, 0, 0]
2	1	9	[0, 0, 1, 0, 0, 1]
3	0	10	[0, 0, 1, 0, 1, 0]
3	1	11	[0, 0, 1, 0, 1, 1]
2	2	12	[0, 0, 1, 1, 0, 0]
2	3	13	[0, 0, 1, 1, 0, 1]
3	2	14	[0, 0, 1, 1, 1, 0]
3	3	15	[0, 0, 1, 1, 1, 1]

Larger Example

10x10 zorder



Larger Example

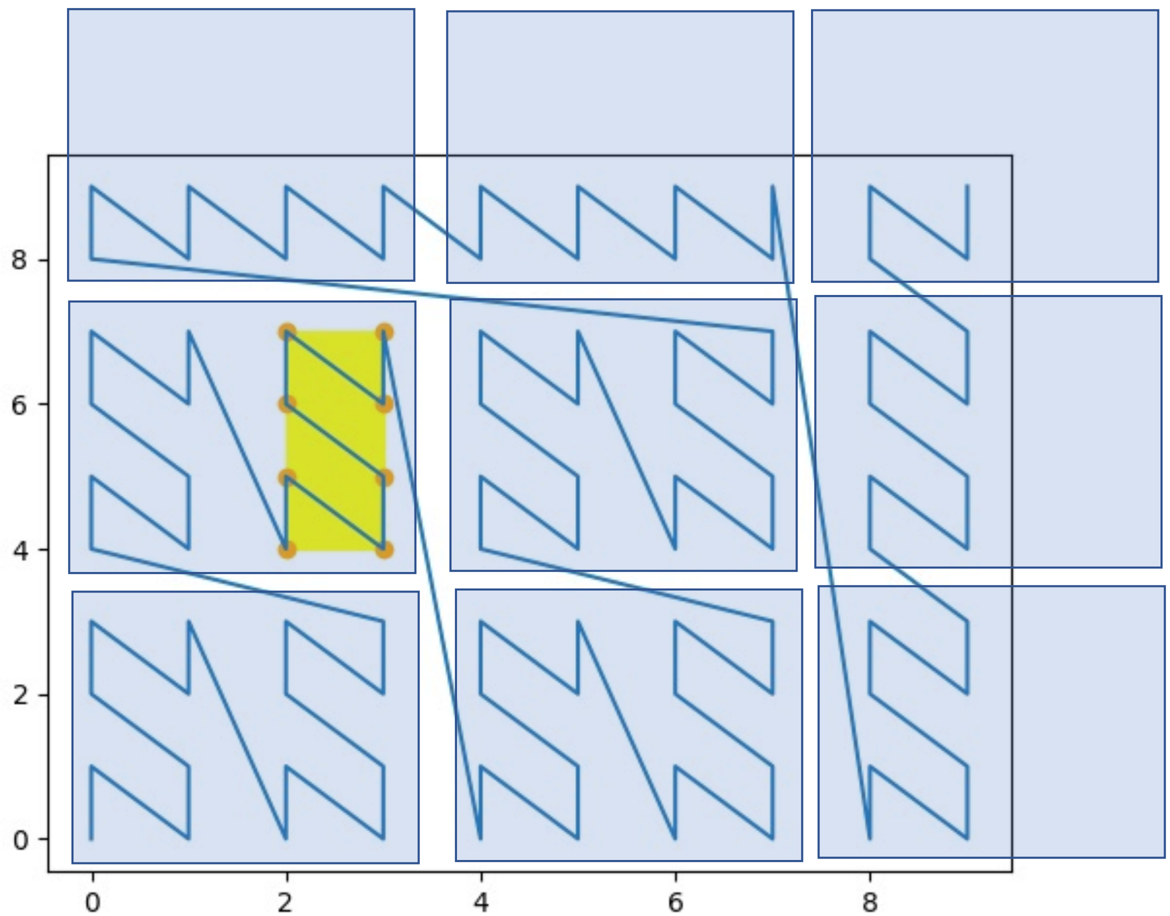
See zorder.py

10x10 zorder

Query from (2,4) to (3,7)

All records in rectangle are contiguous in zorder

Overlaying pages, we can read just one



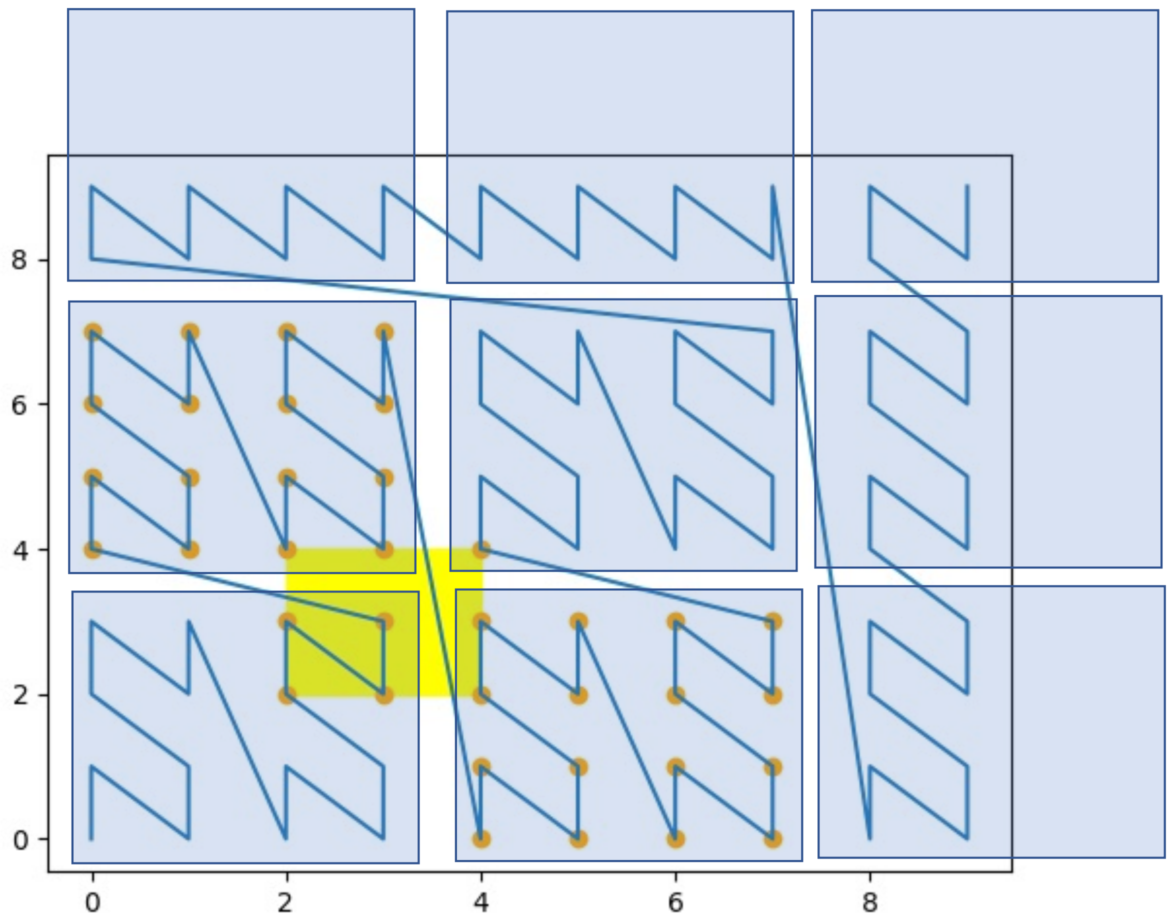
Larger Example

10x10 zorder

Query from (2,2) to (4,4)

9 records in range are

37 records between smallest and largest zorder



Actual wasted I/O depends on page structure

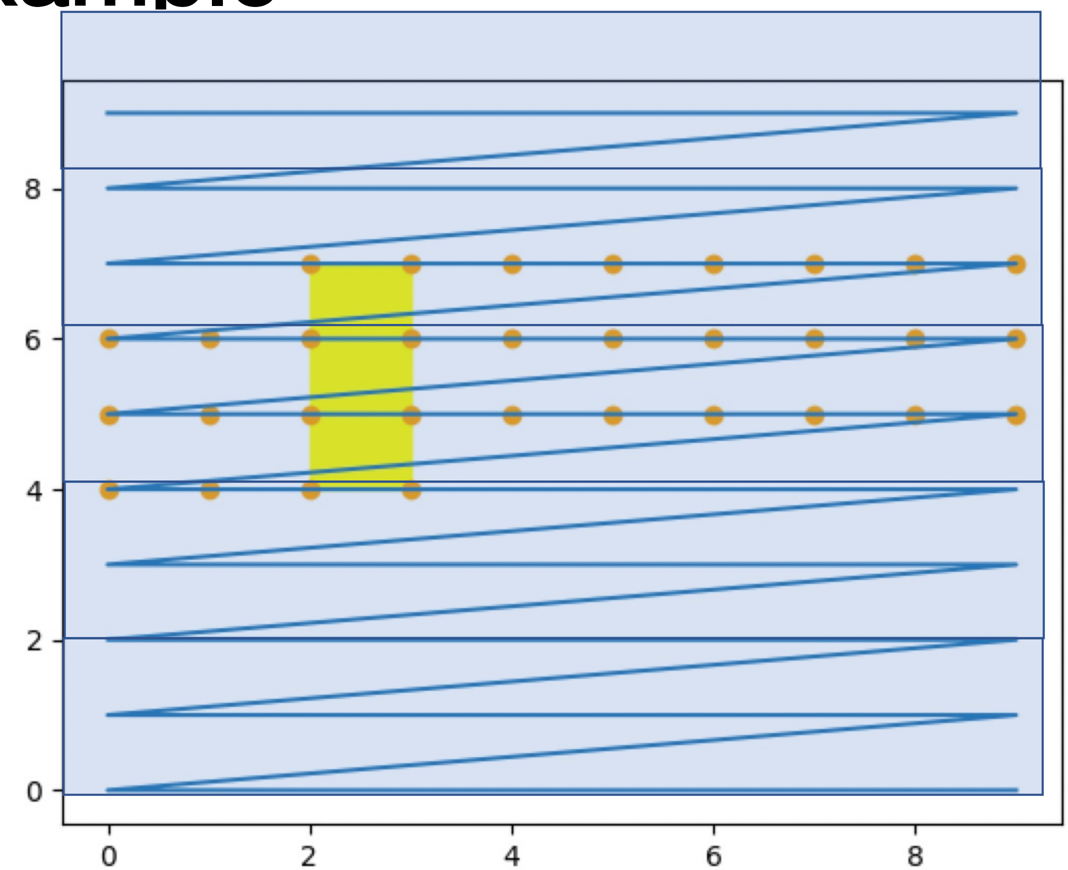
Here we would read 4 pages, with 64 records, 9 of which we need

Row Order Example

8 records in range

32 records between smallest and largest roworder

If split into pages, need to read 3 pages, with 60 records on them, to get 8 records



Clicker Q1

- Table of sales, with sale price, region, date, store, customer, and many other columns
- For each query, which layout would you recommend, if this is the only query your system needs to run

Choose A, B, or C

A) Column store, ordered by date, partitioned region

B) Row store

C) Column store, ordered by price, partitioned by store

SELECT MAX(price) FROM sales GROUP BY store

Clicker Q2

- Table of sales, with sale price, region, date, store, customer, and many other columns
- For each query, which layout would you recommend, if this is the only query your system needs to run

Choose A, B, or C

A) Column store, ordered by date, partitioned region

B) Row store

C) Column store, ordered by price, partitioned by store

INSERT INTO sales VALUES (….)

Clicker Q3

- Table of sales, with sale price, region, date, store, customer, and many other columns
- For each query, which layout would you recommend, if this is the only query your system needs to run

Choose A, B, or C

A) Column store, ordered by date, partitioned region

B) Row store

C) Column store, ordered by price, partitioned by store

SELECT * FROM sales WHERE customerid = 123211

Compression

- Storage is expensive
- System performance is proportional to the amount of data flowing through the system

Compression Methods

- Entropy coding, e.g., gzip, zlib, …
  - General purpose, good overall compression
- Delta encoding
  - Encode differences, e.g., 1, 2, 3, 4 -> 1, +1, +1, +1
- Run length encoding
  - Suppress duplicates, e.g., 2, 2, 2, 3, 4, 4, 4, 4, 4, -> 2x3, 3x1, 4x5
- Bit packing
  - Use fewer bits for short integers
  - Pairs well with delta coding

Good for mostly sorted, numeric data (floats)

Good for mostly sorted ints or categorical data

Good for limited precision data

- Performance vs space tradeoff
- Some compression can be directly operated on, e.g., RLE
- As with sorting, modifying compressed data in place is difficult

Speed / Performance Tradeoff In Entropy Compression Methods

Compressor name	Ratio	Compression	Decompress.
zstd 1.4.5 -1	2.884	500 MB/s	1660 MB/s
zlib 1.2.11 -1	2.743	90 MB/s	400 MB/s
brotli 1.0.7 -0	2.703	400 MB/s	450 MB/s
zstd 1.4.5 --fast=1	2.434	570 MB/s	2200 MB/s
zstd 1.4.5 --fast=3	2.312	640 MB/s	2300 MB/s
quicklz 1.5.0 -1	2.238	560 MB/s	710 MB/s
zstd 1.4.5 --fast=5	2.178	700 MB/s	2420 MB/s
lzo1x 2.10 -1	2.106	690 MB/s	820 MB/s
lz4 1.9.2	2.101	740 MB/s	4530 MB/s
lzf 3.6 -1	2.077	410 MB/s	860 MB/s
snappy 1.1.8	2.073	560 MB/s	1790 MB/s

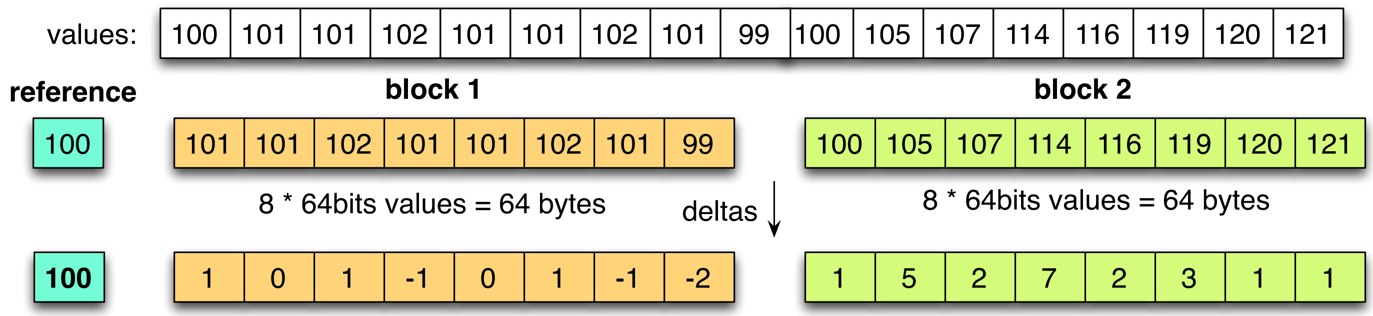
http://facebook.github.io/zstd/

Even 4GB/sec may not be able to keep up with memory!

Compressing a range of text data from the Internet

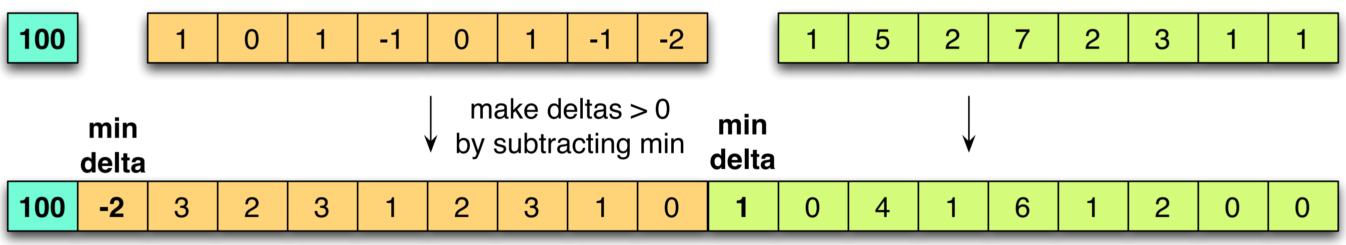
Lightweight schemes will be faster, and less good at text compression, but can do very well for tabular data with few values or regular values

Delta Encoding in Parquet



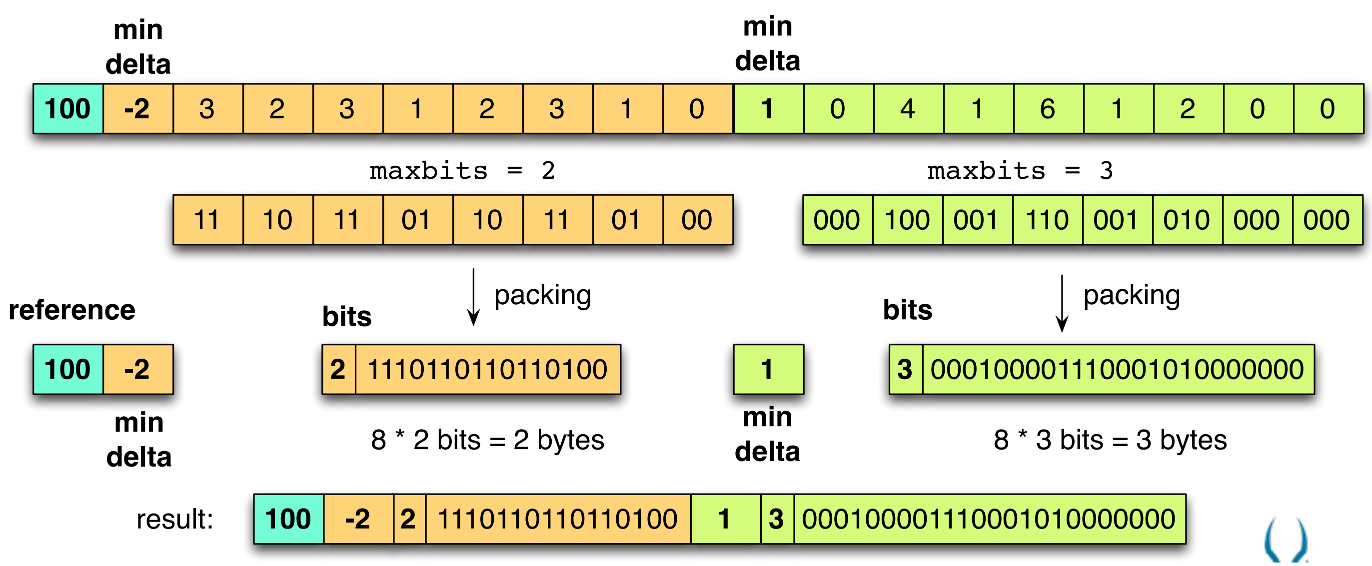
Source "**Efficient Data Storage for Analytics with Apache Parquet 2.0", Julian Le Dem**

Delta Encoding in Parquet



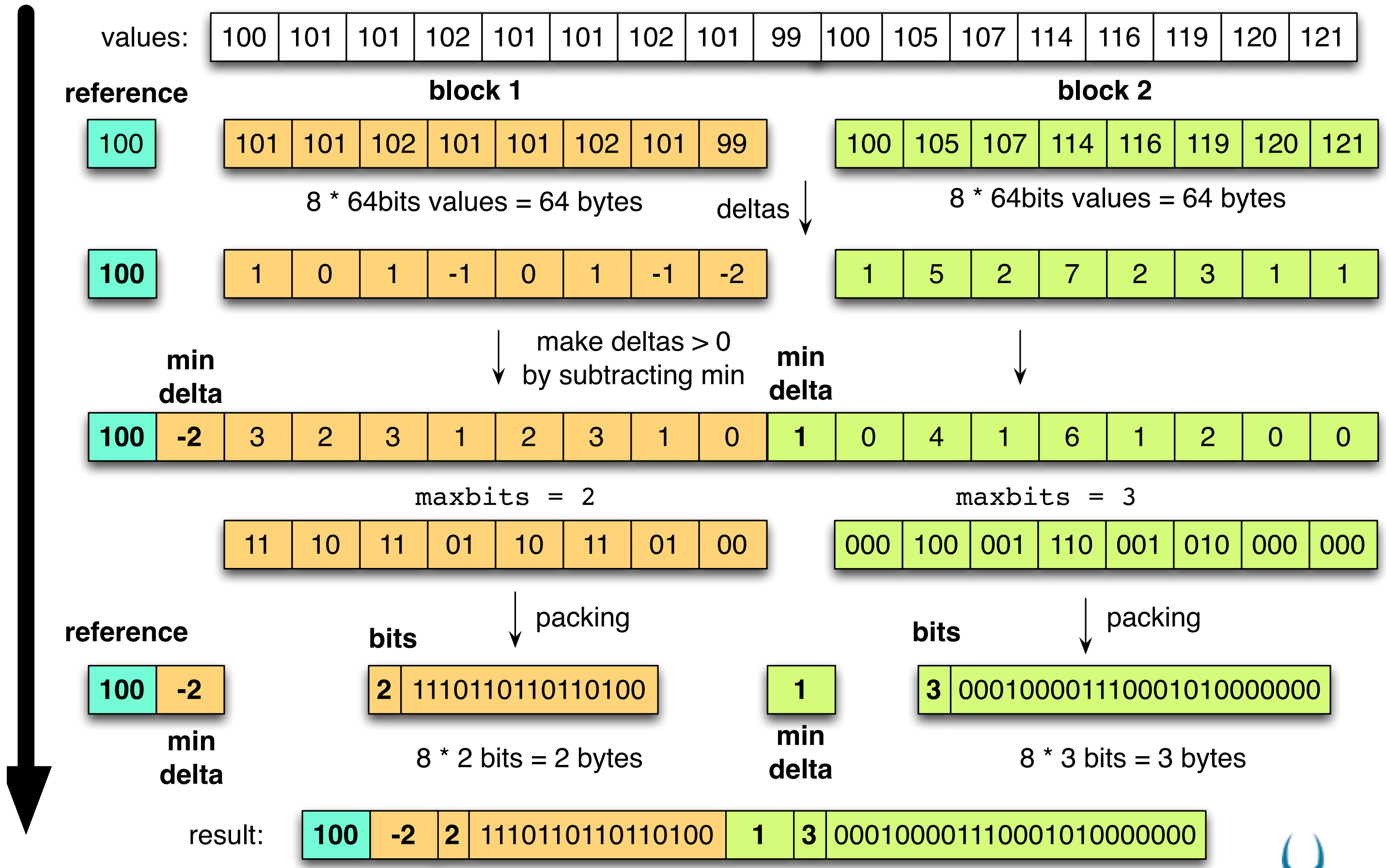
Source “**Efficient Data Storage for Analytics with Apache Parquet 2.0”, Julian Le Dem**

Delta Encoding in Parquet



Source "**Efficient Data Storage for Analytics with Apache Parquet 2.0", Julian Le Dem**

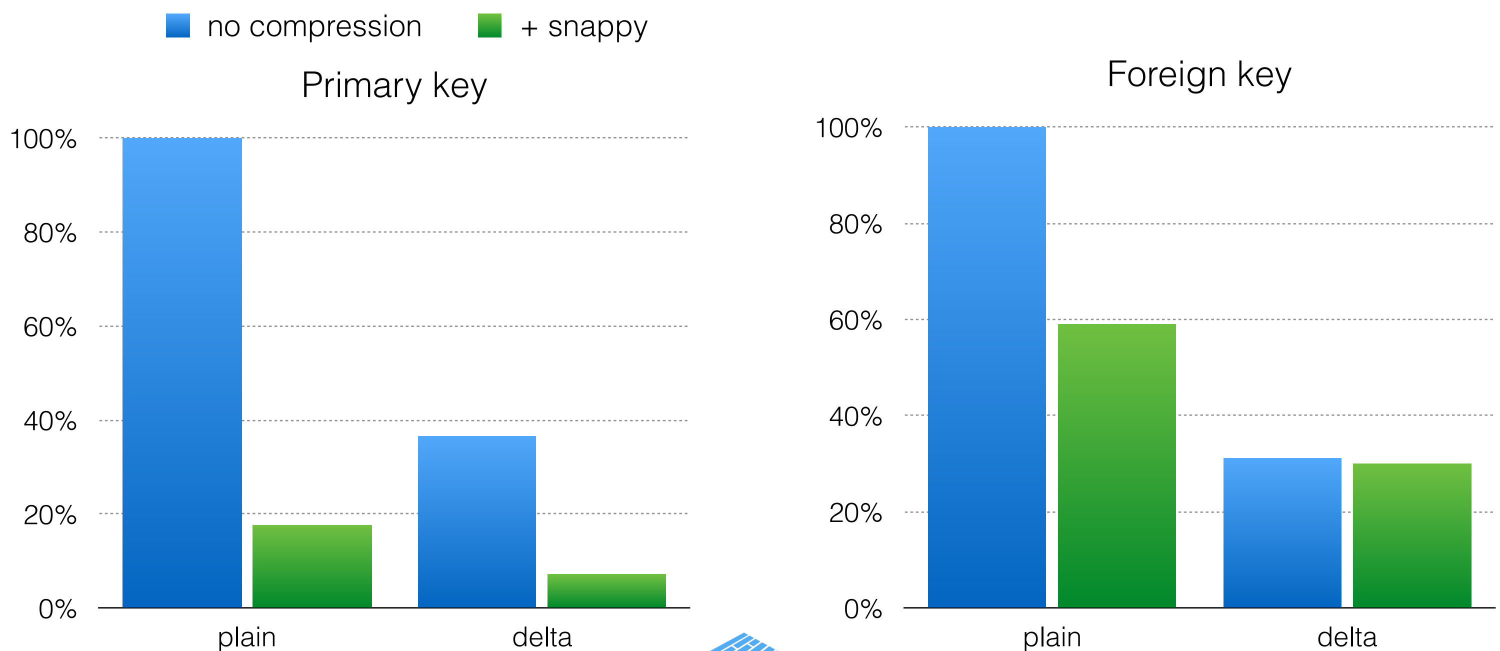
Delta Encoding in Parquet



Source "**Efficient Data Storage for Analytics with Apache Parquet 2.0", Julian Le Dem**

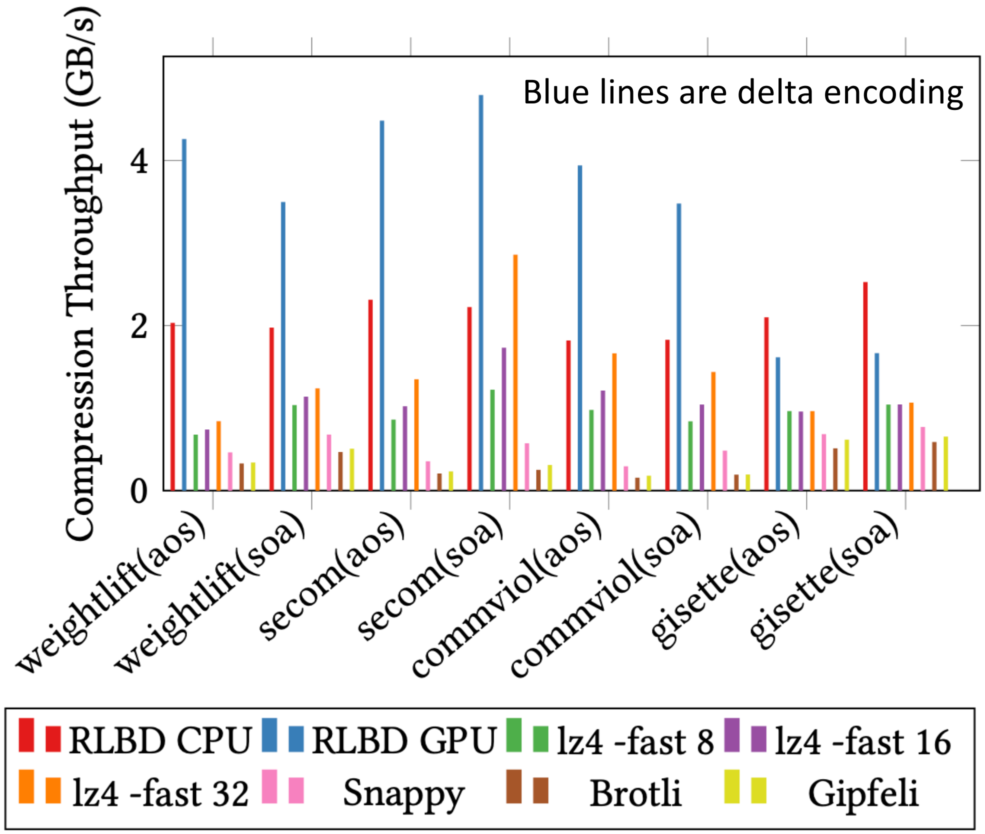
Compression comparison

TPCH: compression of two 64 bits id columns with delta encoding



Source "**Efficient Data Storage for Analytics with Apache Parquet 2.0", Julian Le Dem**

Delta Encoding Can be Very Fast



https://dl.acm.org/doi/10.1145/3229710.3229715

Compression, Con't: Dictionary Encoding

- Dictionary encoding
  - Replace long, frequent values (e.g., strings) with an integer
  - Integer comes from a "dictionary" that maps words to ints
- Reduces data sizes
- Increases access efficiency by eliminating variable size data

Column
Red
Purple
Turquoise
Red
Red
Turquoise
Purple

Encoded Column
1
2
3
1
1
3
2

Dictionary

Val	Decoding
1	Red
2	Purple
3	Turquoise

Compression, Con't: Sparse Data

Table with a lot of NULLs ({})
Arises frequently in ML apps,
e.g., due to one-hot encoding

	A	B	C	D	E	F
1	X	{}	{}	{}	{}	Z
2	{}	{}	{}	{}	{}	Y
3	{}	{}	{}	{}	{}	U
4	{}	{}	{}	K	{}	{}
5	{}	{}	{}	{}	{}	{}

If we represent NULLs as a value, will waste a lot of space

If > X% of data is NULL, store data as a list of non-null tuples, e.g.:

1A: X, 1F: Z, 2F: Y, 3F:U, 4D: K

Need to store row/column identifiers explicitly, but can be much more compact

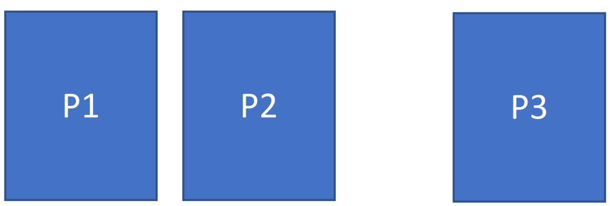
Handling New Data

- In most data science applications, we don't update existing data
- Do need need to deal with new data that is arriving
- If we have a complex data layout, e.g., sorted, partitioned, columns, inserting that data will be slow, because we'll have to rewrite all data
- Idea: just create a new partition for new data, and write your program to merge results from all partitions

Problem: Lots of Partitions

- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times



Problem: Lots of Partitions

- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times

Problem: Lots of Partitions

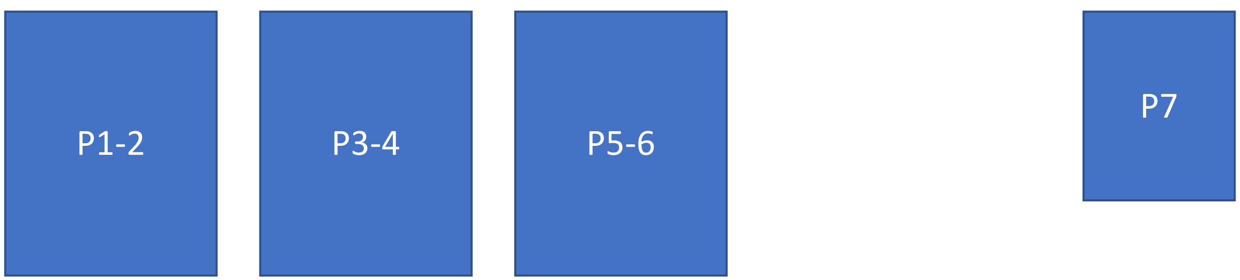
- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times

Problem: Lots of Partitions

- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

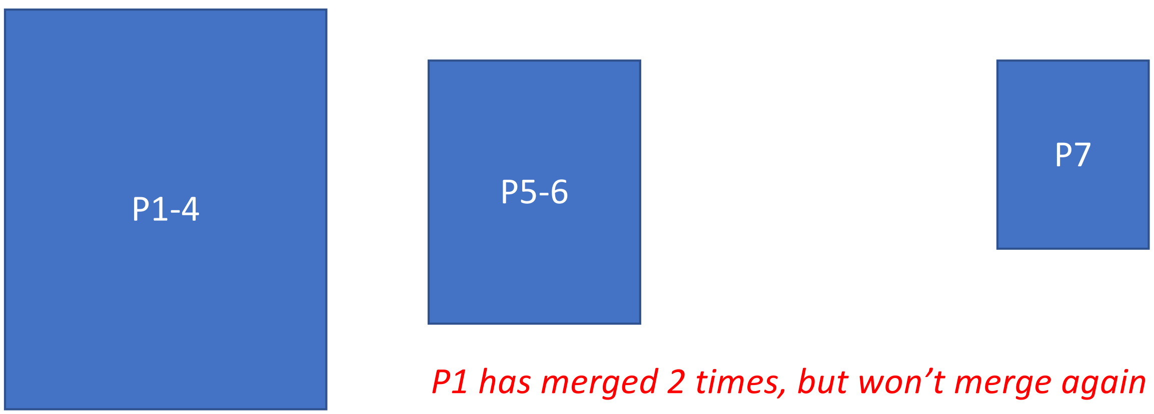
- Log structured merge tree: arrange so partitions merge a logarithmic number of times



Problem: Lots of Partitions

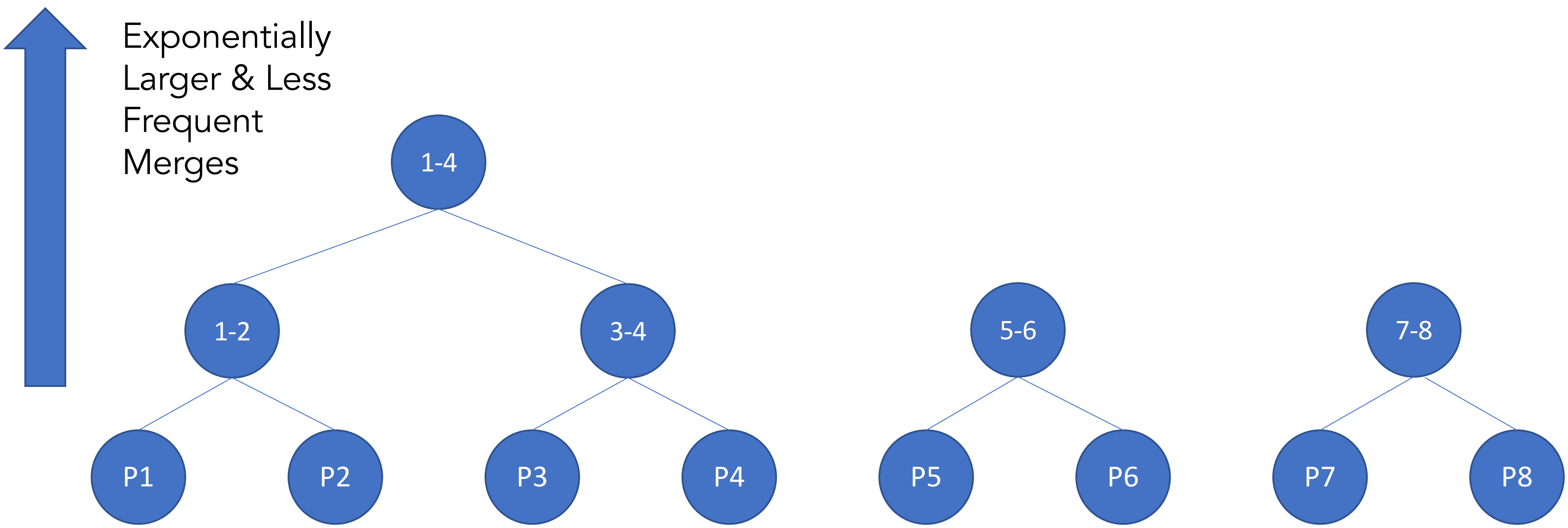
- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times



P1 has merged 2 times, but won't merge again until after 8 more partitions arrive

Log Structure Merge Tree



Summary

- Proper data layouts can dramatically increase performance of data accesses
- Looked at many variations:
  - Column vs row-orientation
  - Multidimensional layouts
    - Quad trees
    - Z-Order
  - Compression
  - Log-structured merging