# Data Layouts

6.S079 Lecture 16

Sam Madden

4/6/2022

# Last Time: Performance

- Python vs pandas vs C vs SQL
- Quantifying performance: bandwith, latency, etc
- Finding & fixing performance issues
- Indexing & join algorithms

# This Time: Data Layouts

- Key ideas:
  - Data Locality
  - Horizontal and Vertical Partitioning
  - Multi-dimensional Layouts
  - Compression
  - Sparse Data
  - Log-structured Merge Trees

# 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%'

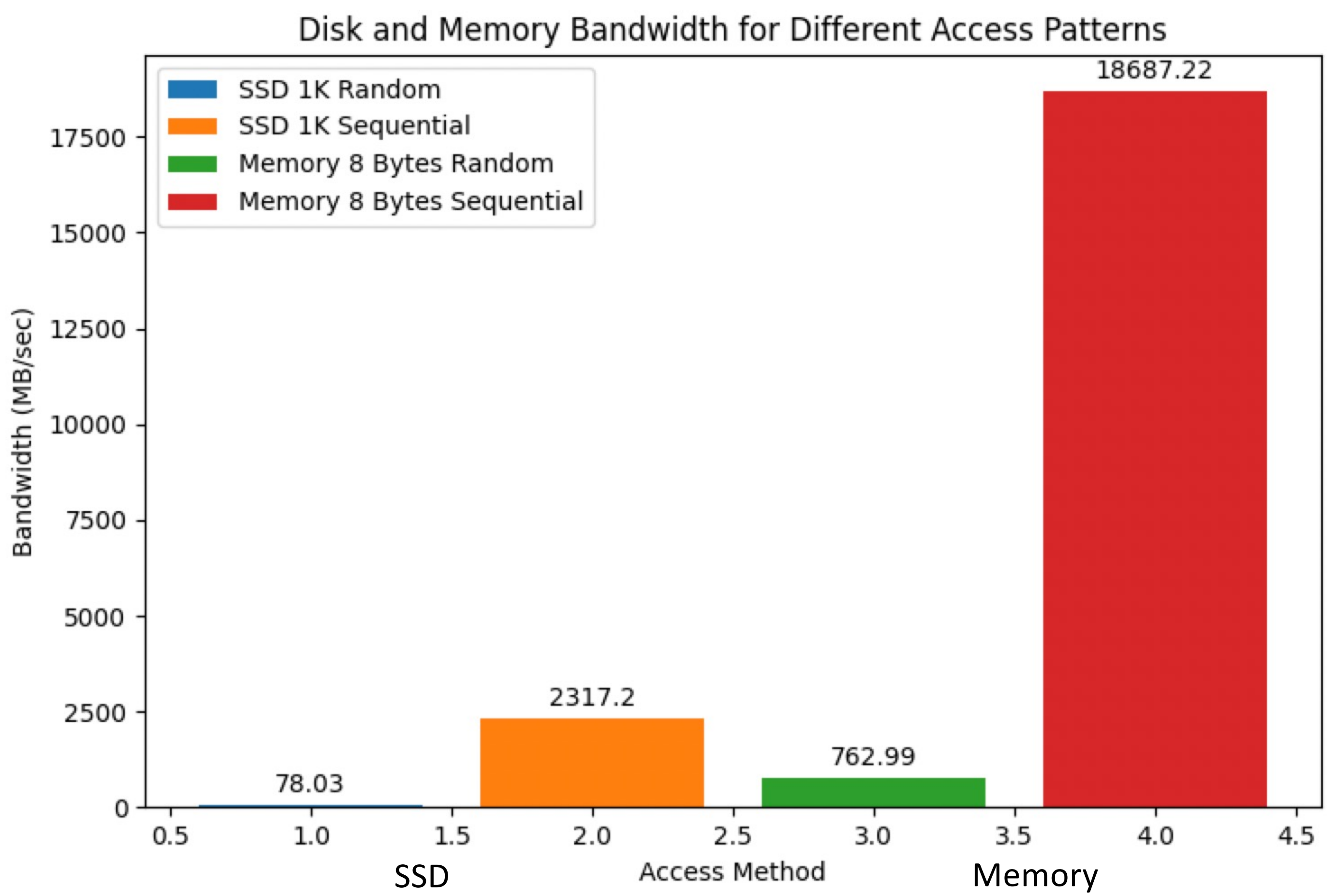
*Sorted in name order*
*All "MAD" records on same few disk/memory blocks ➔ Sequential access to just those blocks*

...
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

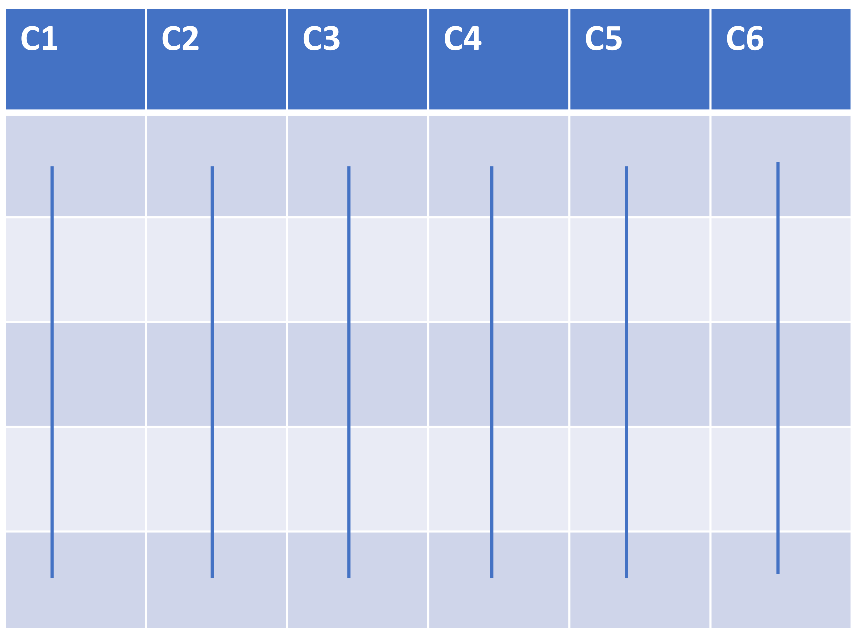
# Linearizing a Table – Row store

| 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"

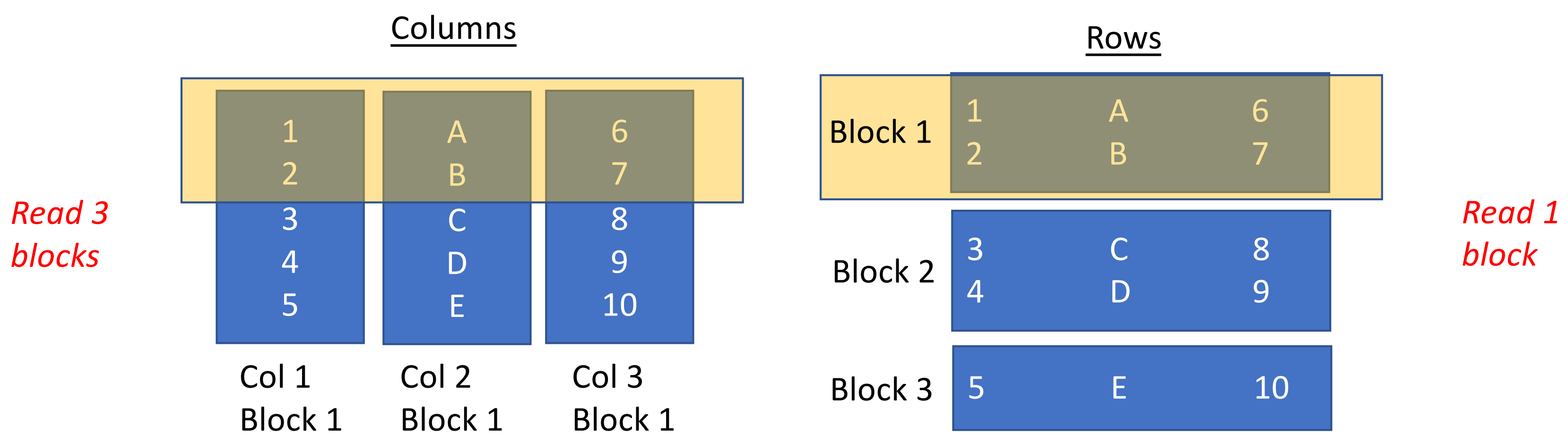



Memory/Disk
(Linear Array)

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?

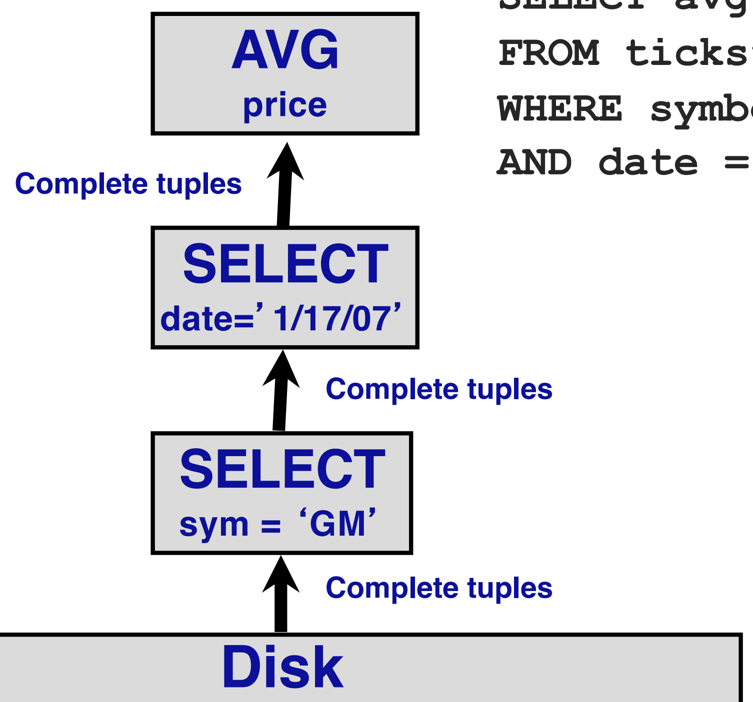
- When only a subset of columns need to be accessed
- When looking at many records
- Reading data from N columns of a few column-oriented records may be *worse* than using a row-oriented representation



# Query Processing Example

- Traditional Row Store

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

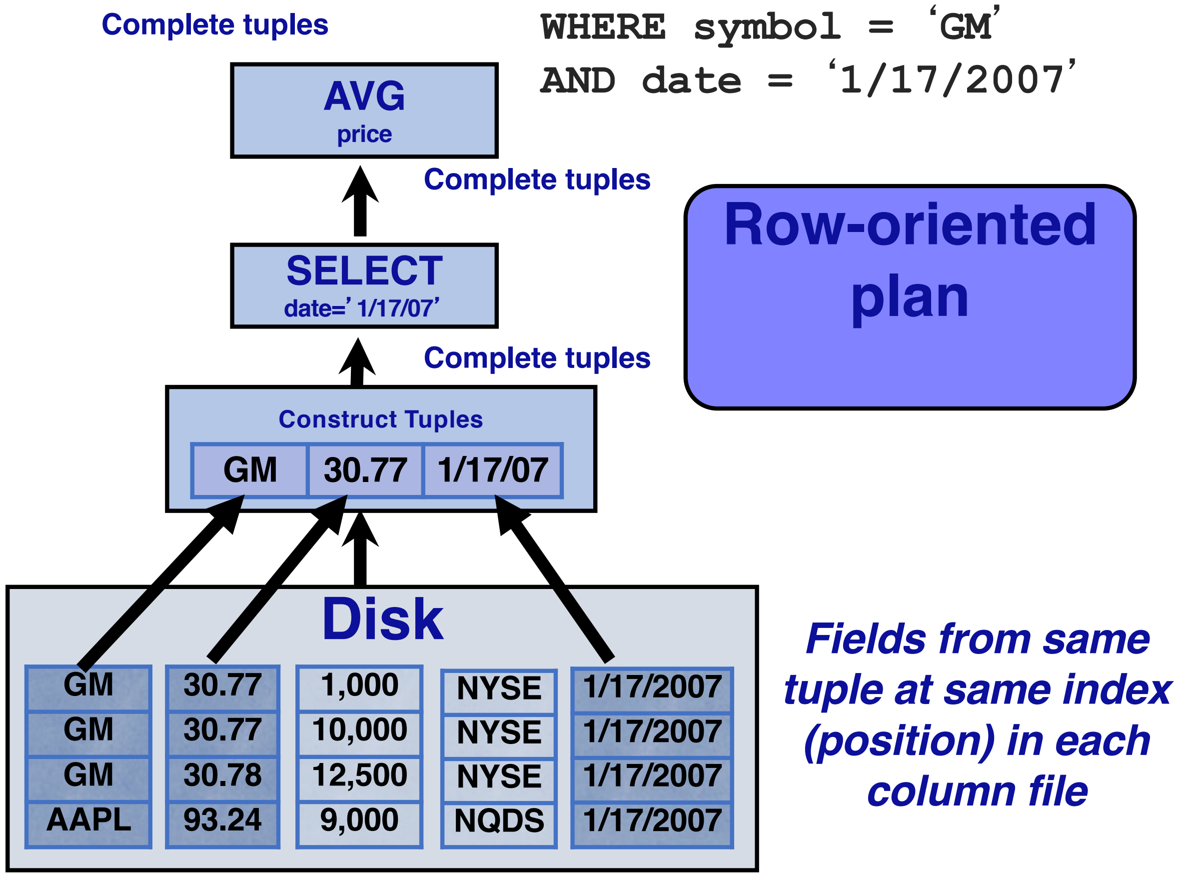



| | | | | |
|---|---|---|---|---|
| GM | 30.77 | 1,000 | NYSE | 1/17/2007 |
| GM | 30.77 | 10,000 | NYSE | 1/17/2007 |
| GM | 30.78 | 12,500 | NYSE | 1/17/2007 |
| AAPL | 93.24 | 9,000 | NQDS | 1/17/2007 |

# Query Processing Example

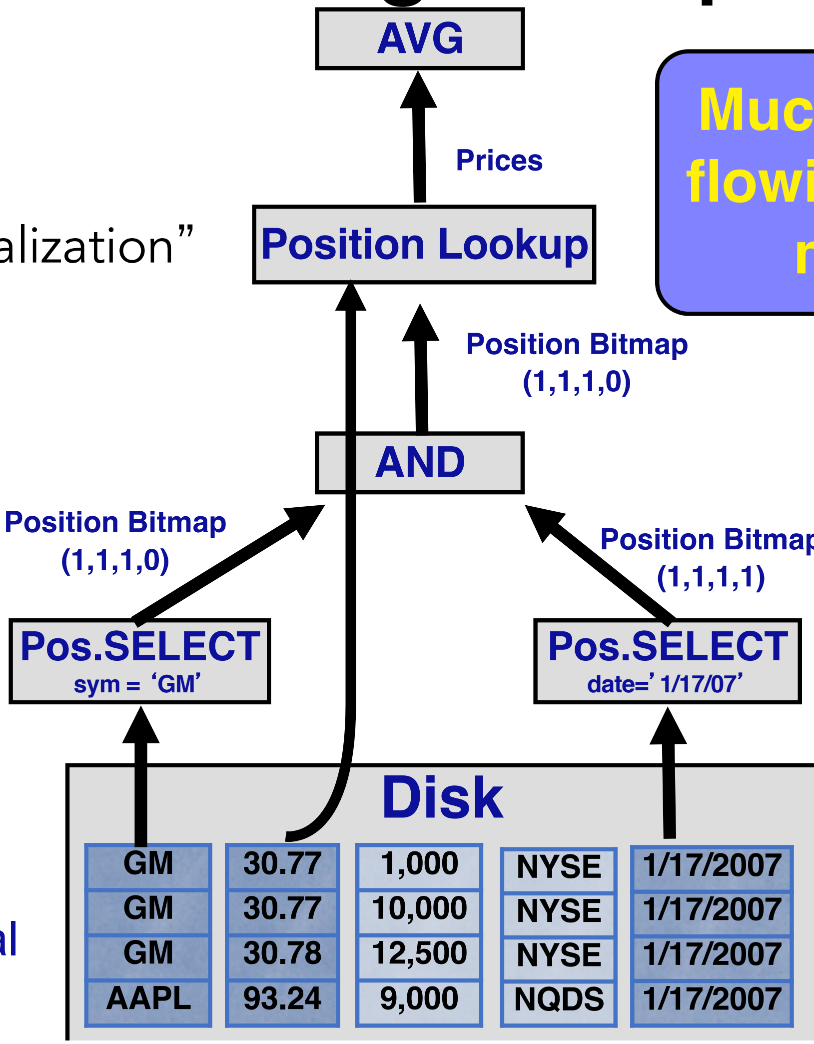
- Basic Column  Store
- "Early Materialization"

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



# 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

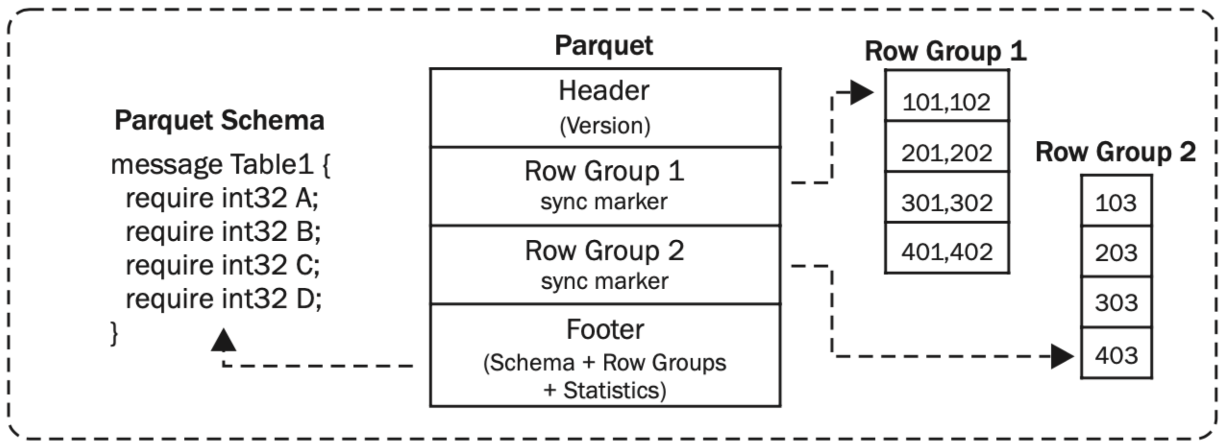
- Parquet is a column-oriented data form for storing tabular data

- 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

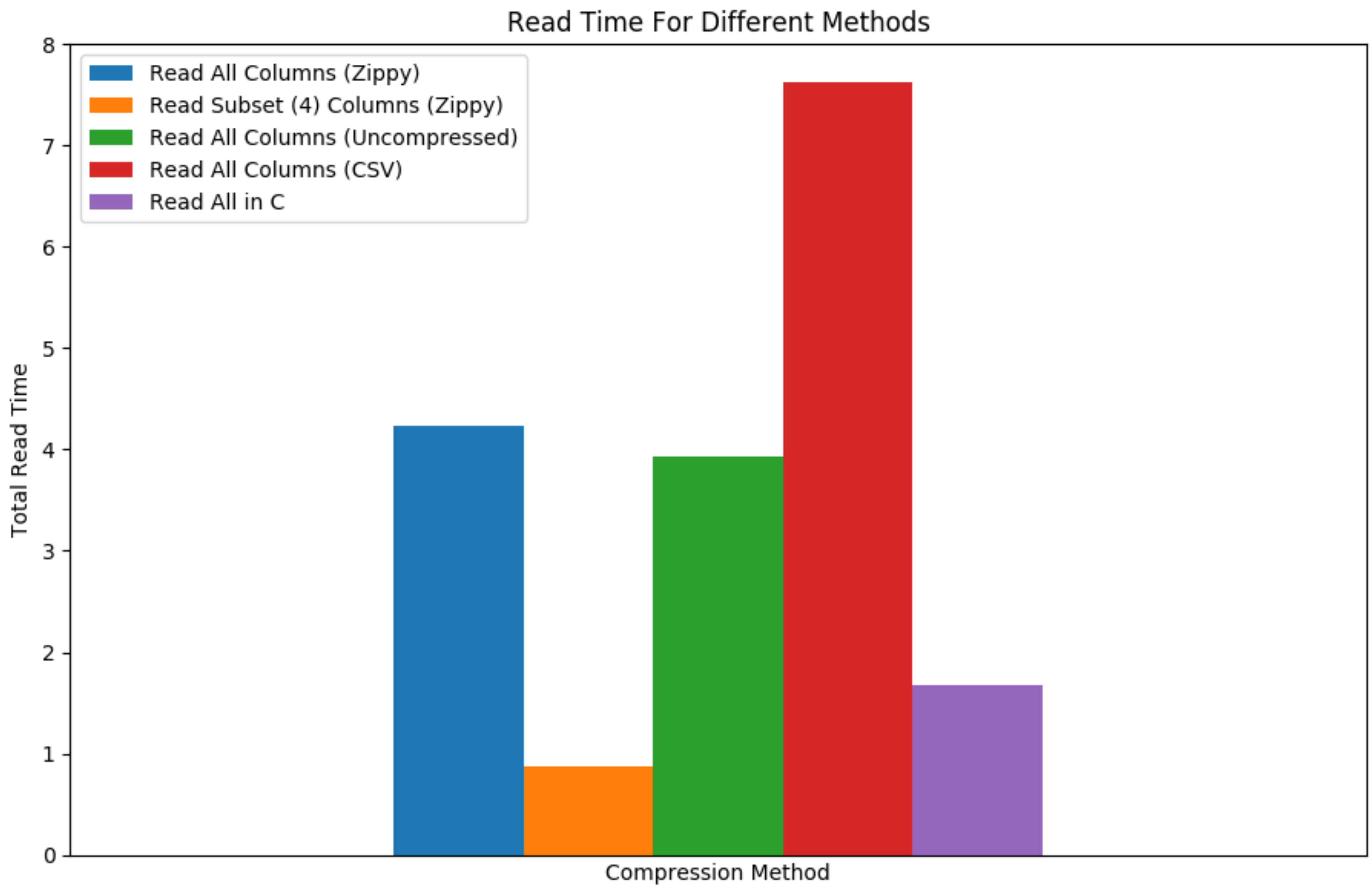
Table 1

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

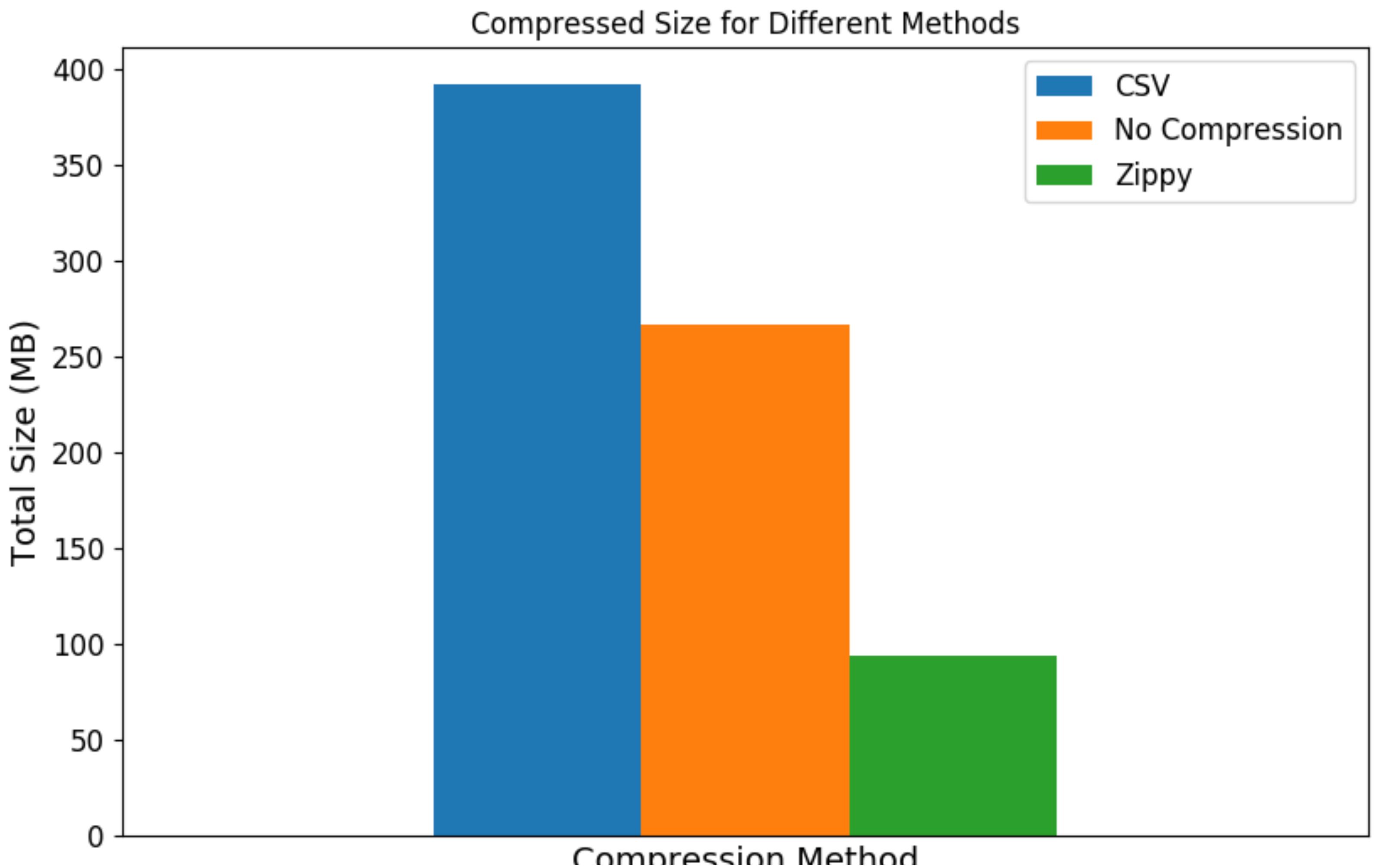



From “A Cost-based Storage Format Selector for Materialization in Big Data Frameworks”, Faisal et al

# 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 | |

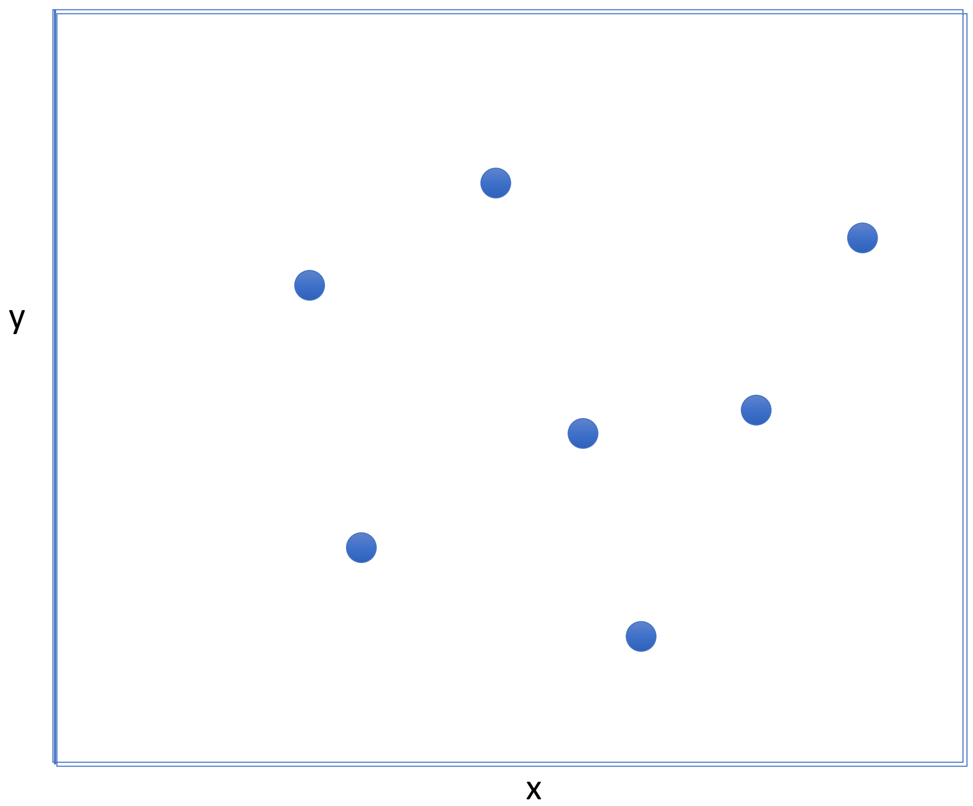
| Date | Region | Profit |
|---|---|---|
| 1/3/2019 | NE | |
| 1/3/2019 | SE | |
| 1/3/2019 | SW | |

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

# What if I want to partition on several attributes?

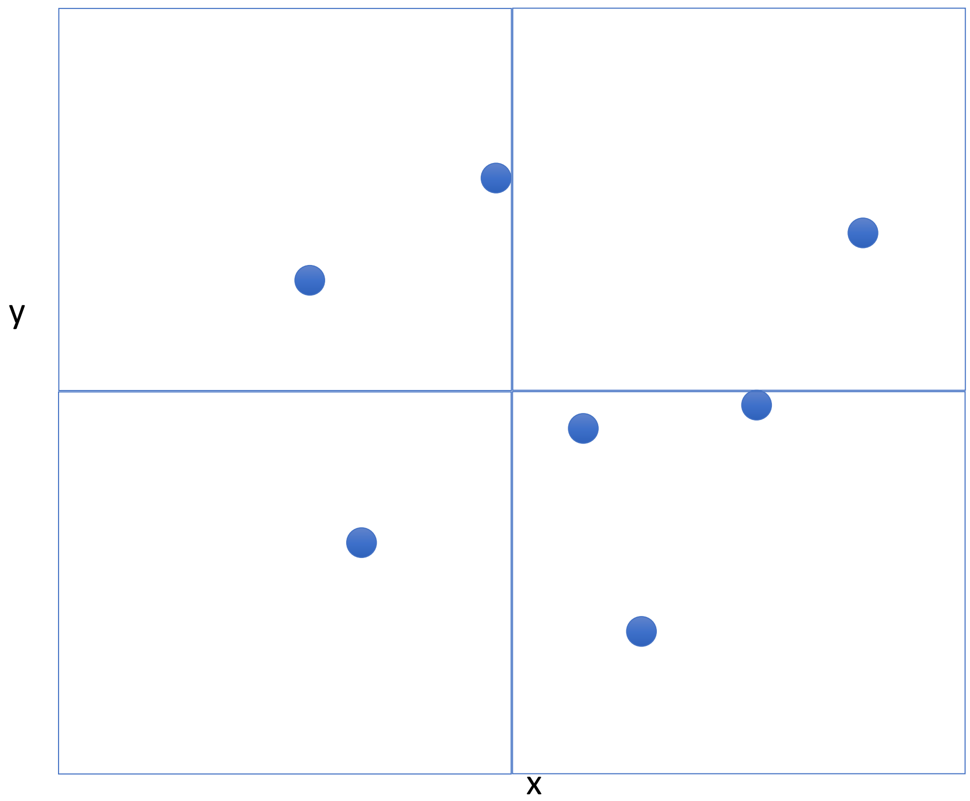
- Basic idea: ”tile” data into N dimesions
- 2 approaches:
- **Quad-tree:** recursively subdivide until tiles are under a target size
- **Z-order**: interleave multiple dimensions, order by interleaving

# Quad-Tree



# Quad-Tree

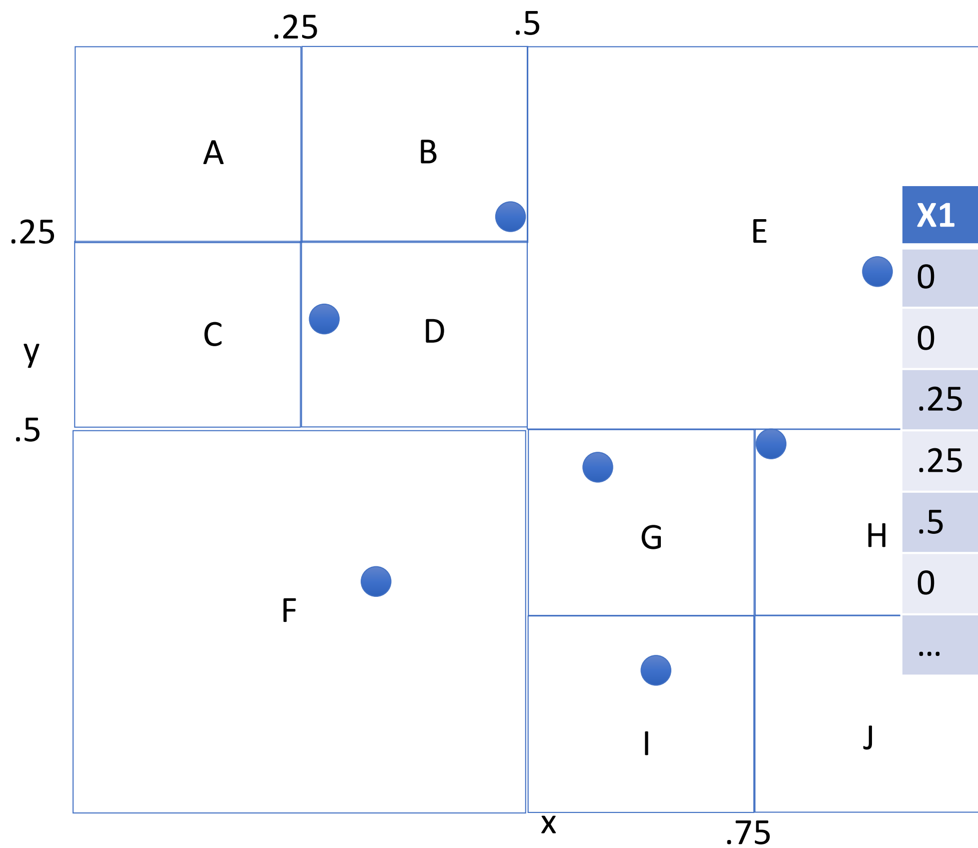
Recursively subdivide



# Quad-Tree

Until partitions are of some maximum size

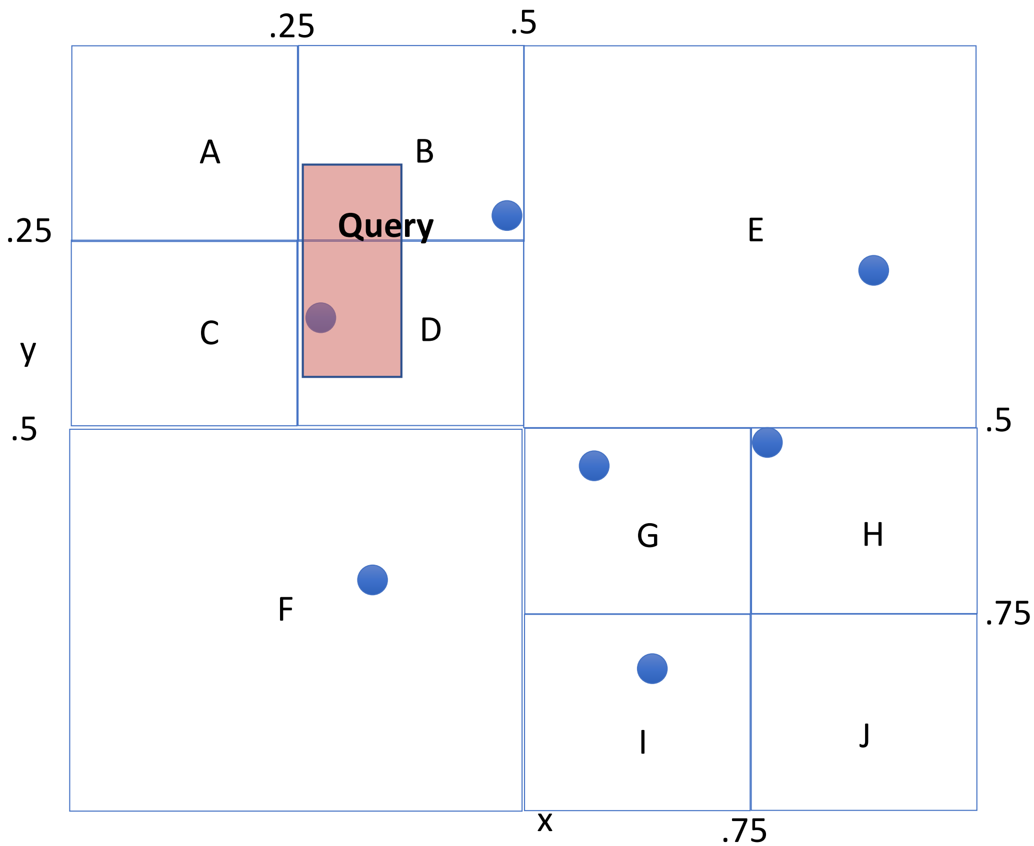
Index stores boundaries of rectangles, and pointers on disk




| 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

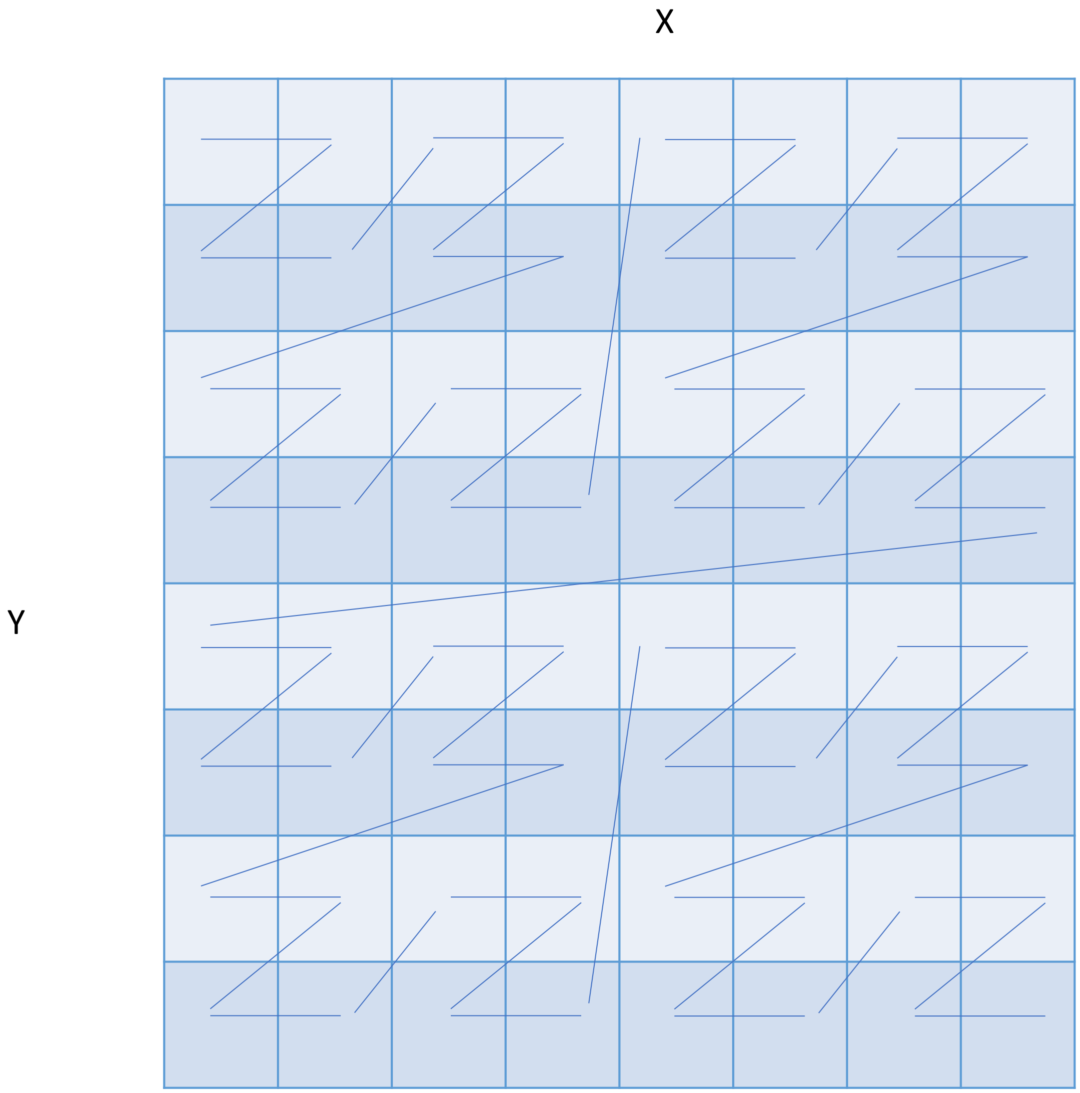
Until partitions are of some maximum size




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

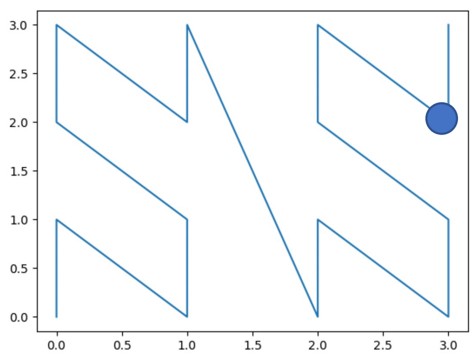
- To generate a Zorder, interleave bits of numbers

e.g., Zorder(3,2)

3 = 0011

2 = 0010

➔ 00001110 = 14

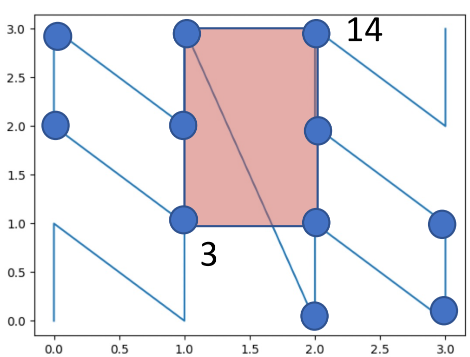



| i | j | zorder | bits |
|---|---|---|---|
| 0 | 0 | 0 | [0, 0, 0, 0, 0, 0] |
| 0 | 1 | 1 | [0, 0, 0, 0, 0, 1] |
| 1 | 0 | 2 | [0, 0, 0, 0, 1, 0] |
| 1 | 1 | 3 | [0, 0, 0, 0, 1, 1] |
| 0 | 2 | 4 | [0, 0, 0, 1, 0, 0] |
| 0 | 3 | 5 | [0, 0, 0, 1, 0, 1] |
| 1 | 2 | 6 | [0, 0, 0, 1, 1, 0] |
| 1 | 3 | 7 | [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))

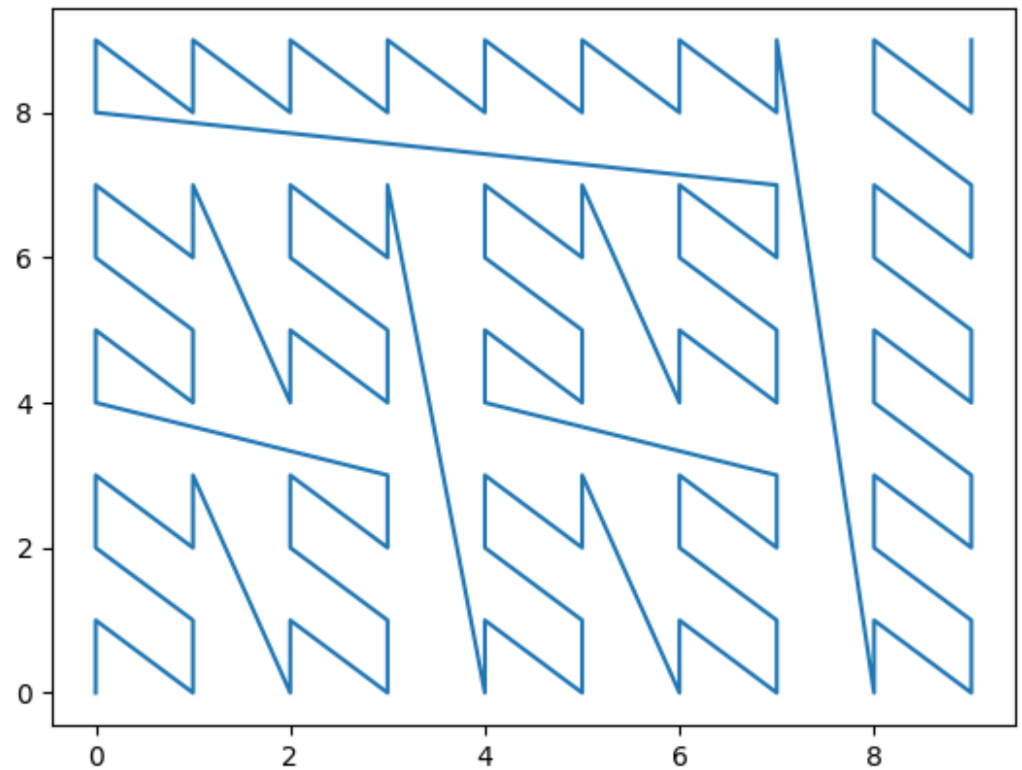
Zorder(1,1) = 0011 = 3
Zorder(2,3) = 1101 = 13




| 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

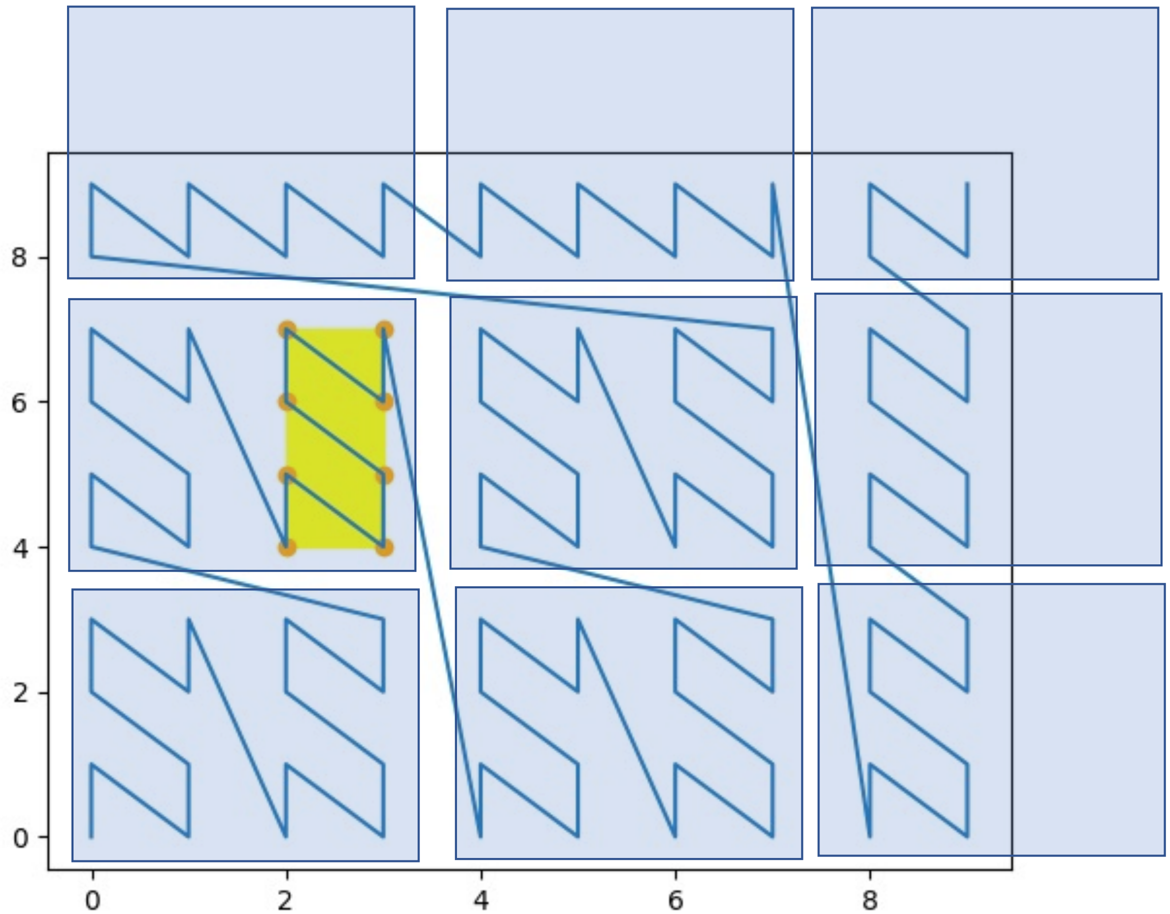
See zorder.py

10x10 zorder

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

All records in rectangle are contiguous in zorder

Overlaying pages, we can read just one



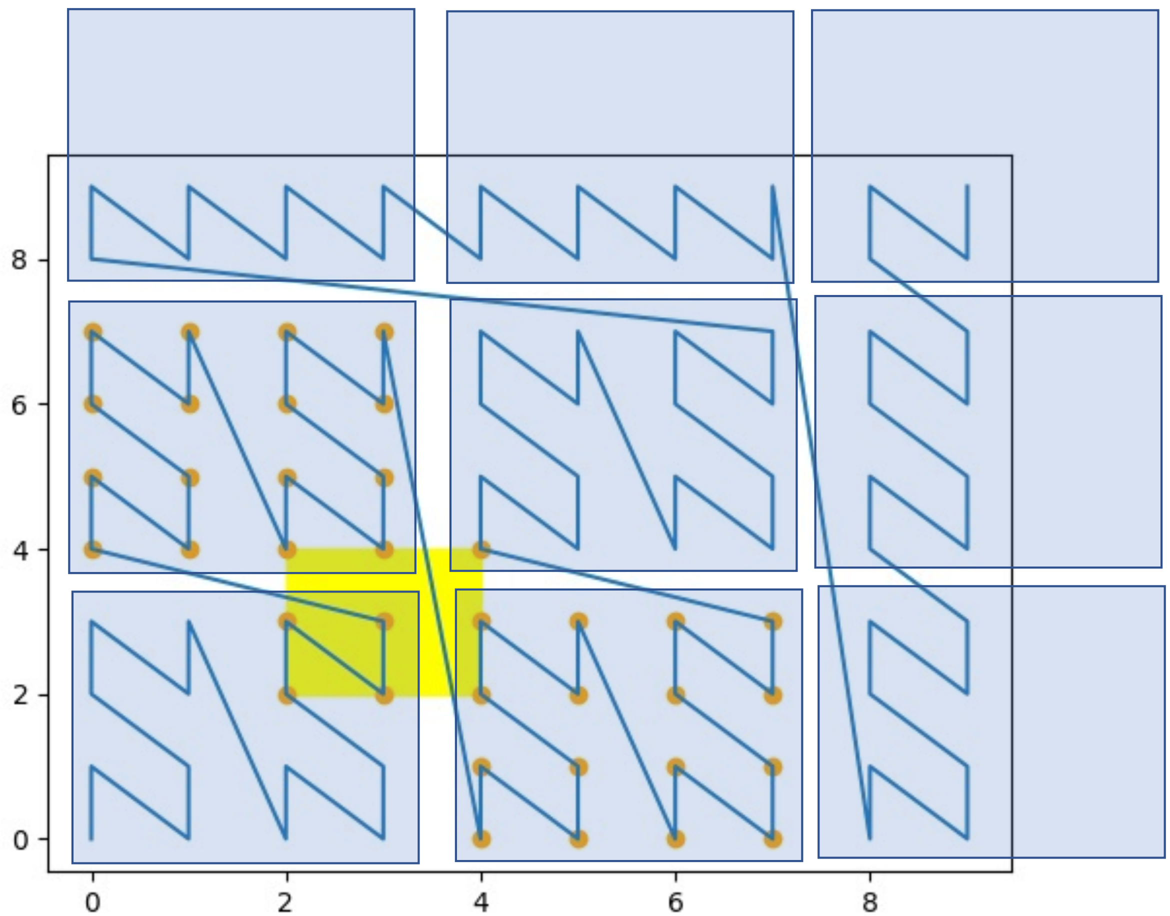
# Larger Example

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

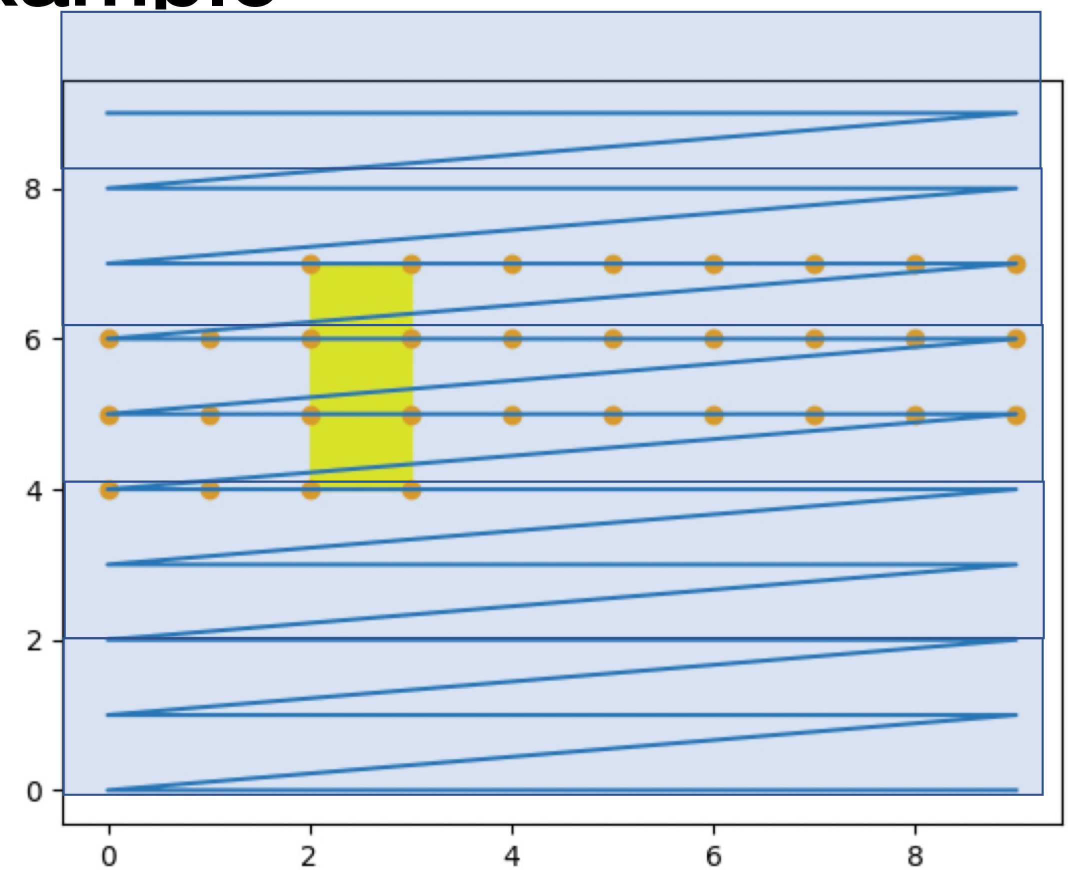
Here we would read 4 pages, with 64 records, 9 of which we need

# 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 |

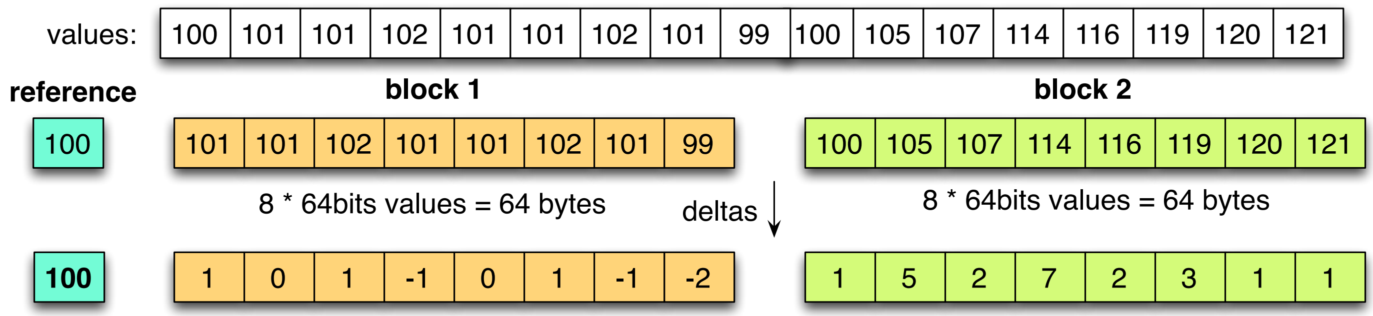
http://facebook.github.io/zstd/

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

Compressing a range of text data from the Internet

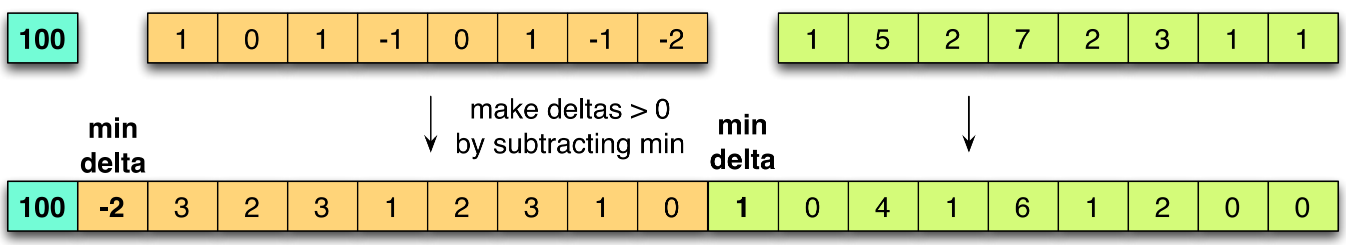
Lightweight schemes will be faster, and less good at text compression, but can do very well for tabular data with few values or regular values

# Delta Encoding in Parquet



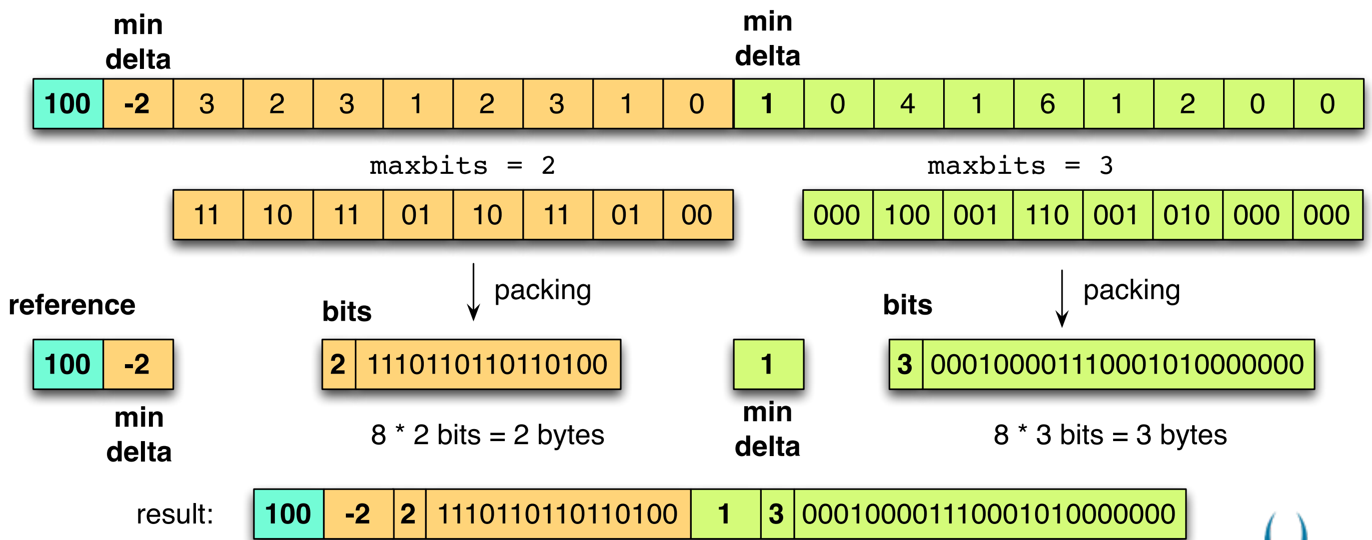

Source "**Efficient Data Storage for Analytics with Apache Parquet 2.0", Julian Le Dem**

# Delta Encoding in Parquet



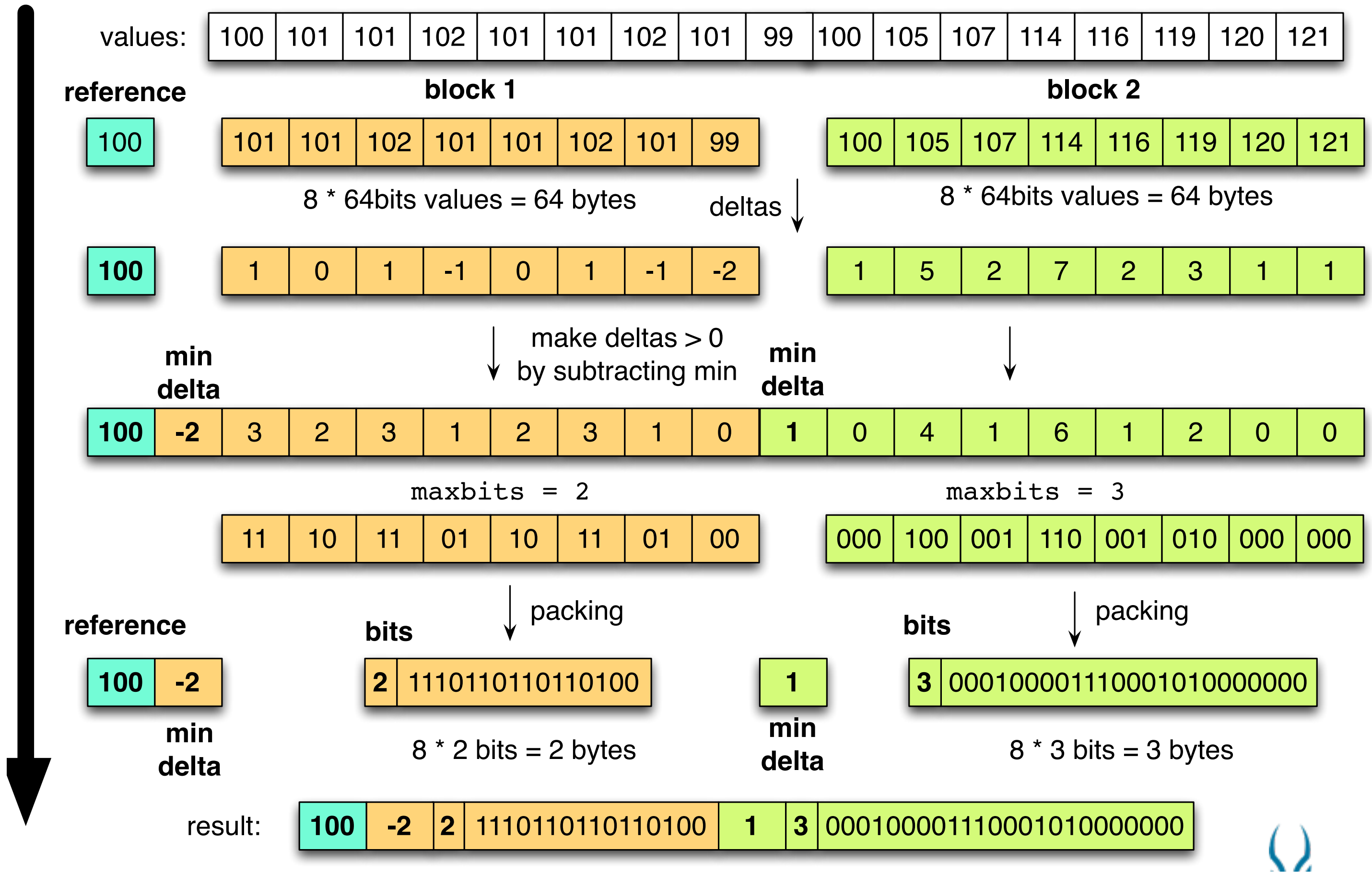

Source "**Efficient Data Storage for Analytics with Apache Parquet 2.0", Julian Le Dem**

# Delta Encoding in Parquet




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

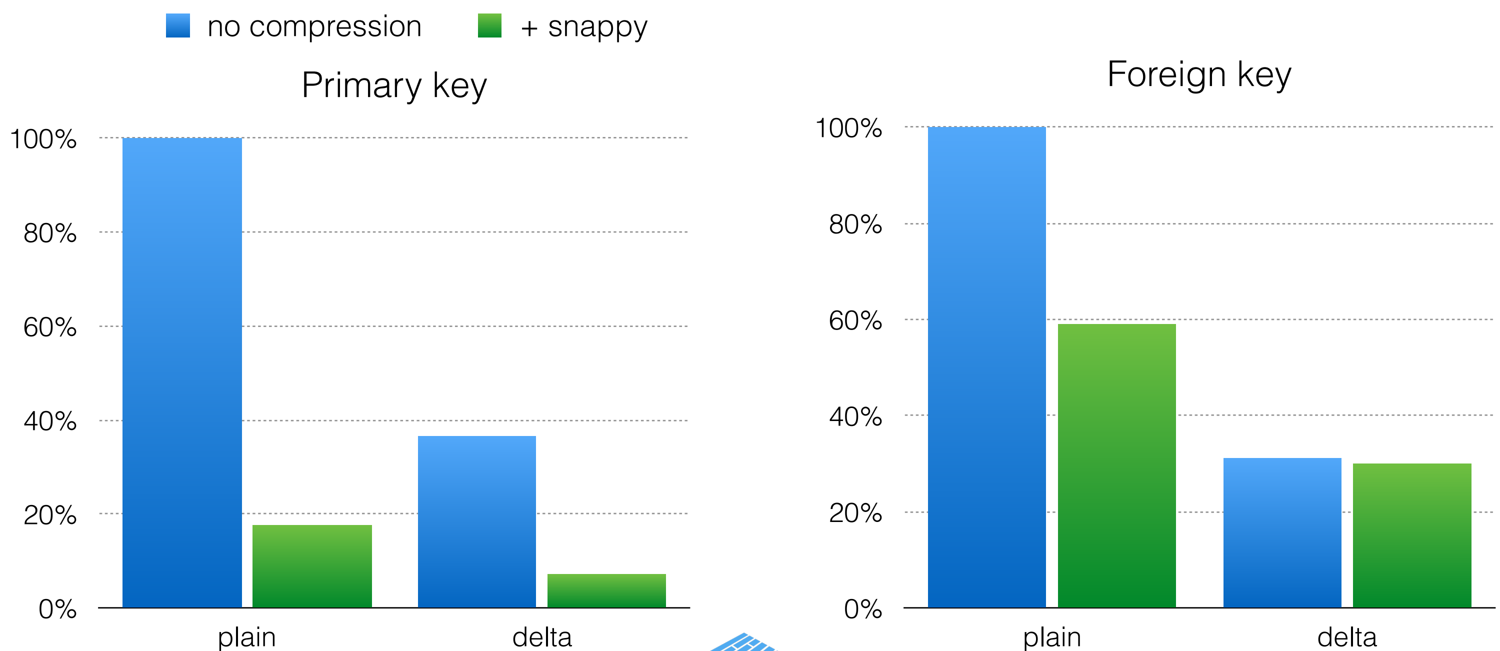
# Delta Encoding in Parquet




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

# Compression comparison

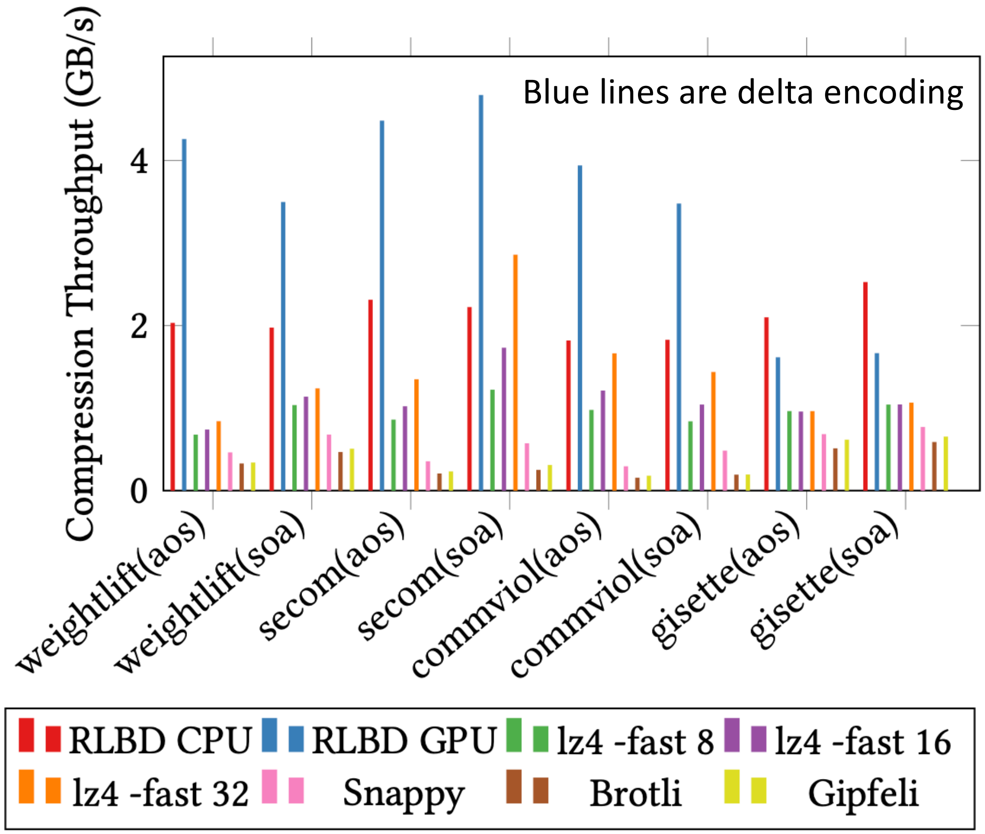
## TPCH: compression of two 64 bits id columns with delta encoding






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

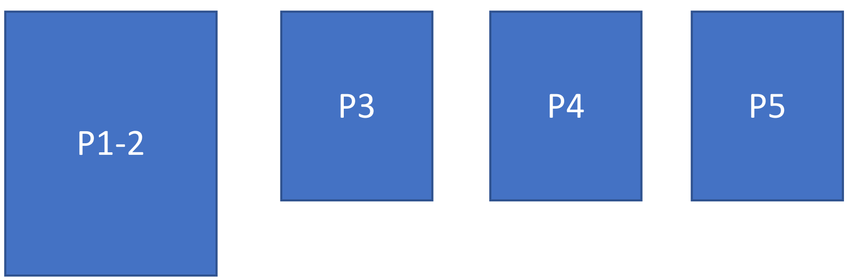
- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- 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



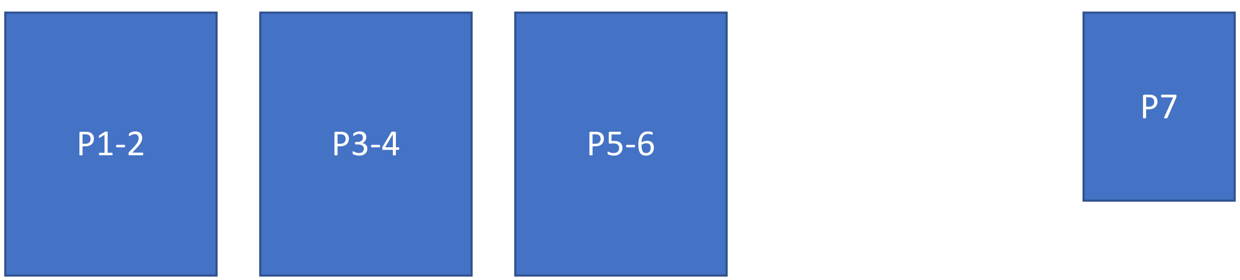
# Problem: Lots of Partitions

- 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

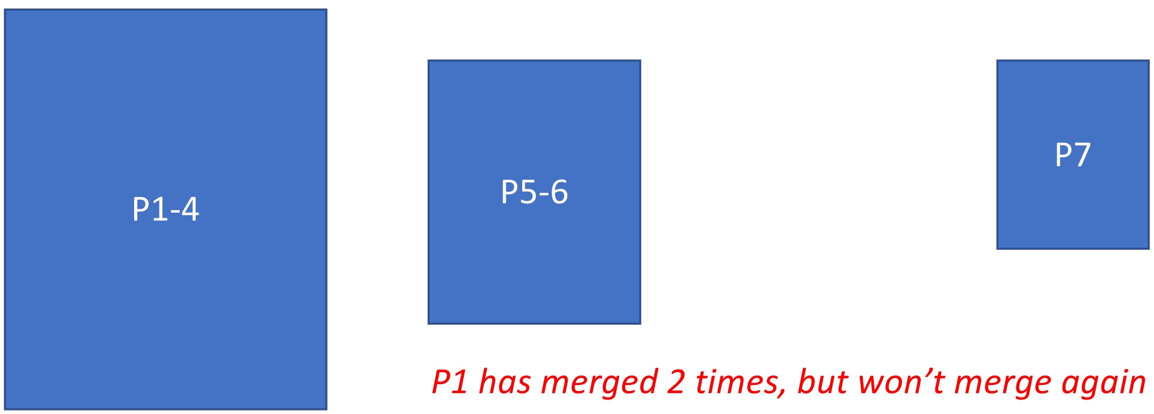
- Performance will degrade as you get many partitions
- Idea:  merge some partitions together, but how?
- Log structured merge tree:  arrange so partitions merge a logarithmic number of times



# Problem: Lots of Partitions

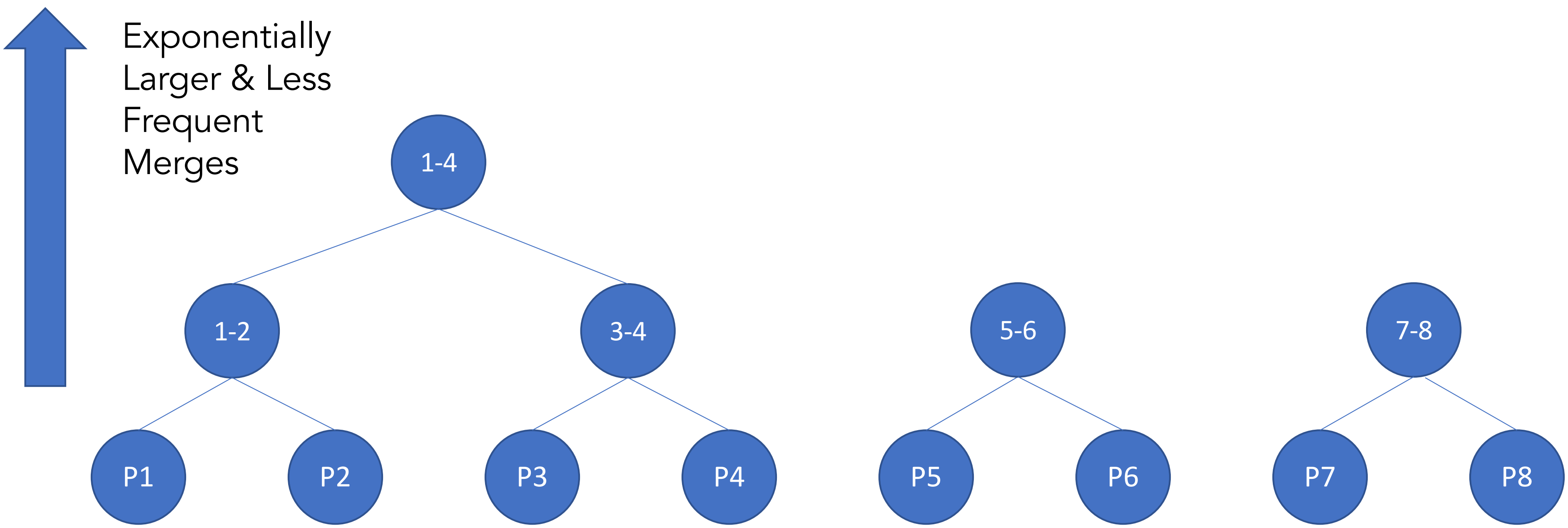
- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- 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