

# Physical Data Organization

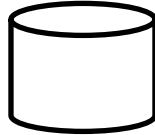
Introduction to Databases

CompSci 316 Spring 2019

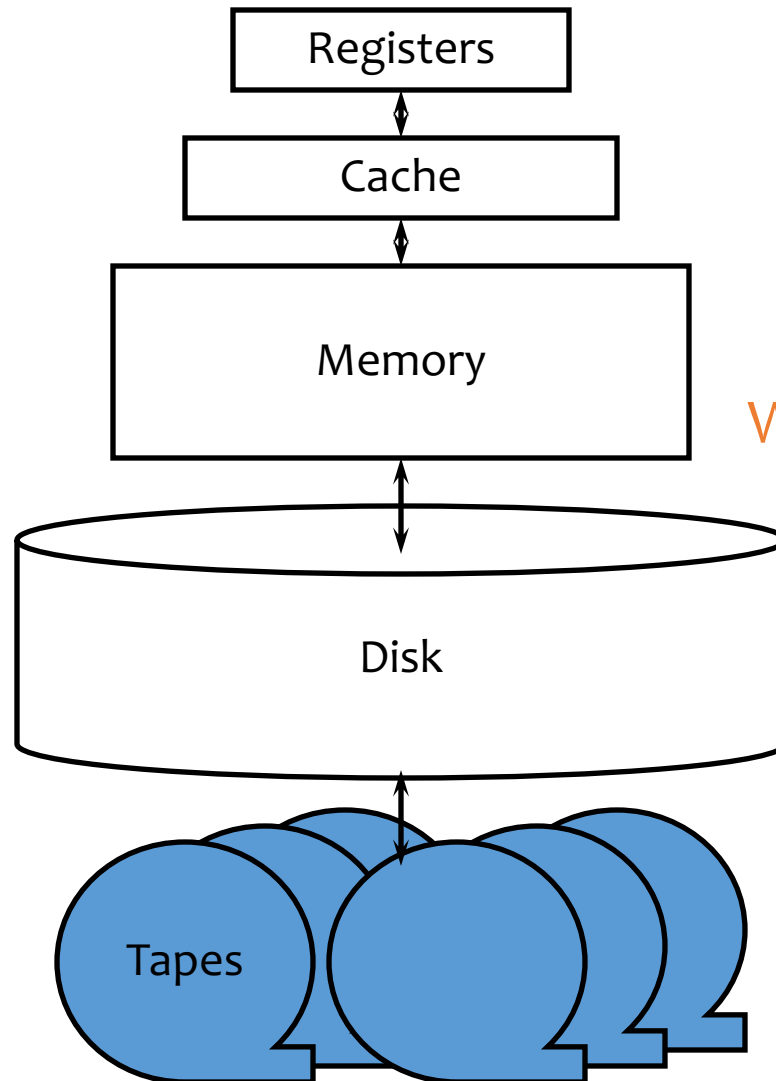


**DUKE**  
COMPUTER SCIENCE

# Outline

- It's all about disks!
  - That's why we always draw databases as 
  - And why the single most important metric in database processing is (oftentimes) the number of disk I/O's performed
- Storing data on a disk
  - Record layout
  - Block layout
  - Column stores

# Storage hierarchy



Why a hierarchy?

# How far away is data?

Location	Cycles	Location	Time
Registers	1	My head	1 min.
On-chip cache	2	This room	2 min.
On-board cache	10	Duke campus	10 min.
Memory	100	Washington D.C.	1.5 hr.
Disk	$10^6$	Pluto	2 yr.
Tape	$10^9$	Andromeda	2000 yr.

(Source: AlphaSort paper, 1995)  
The gap has been widening!

👉 I/O dominates—design your algorithms to reduce I/O!

# Latency Numbers

## Every Programmer Should Know

### Latency Comparison Numbers

-----					
L1 cache reference	0.5	ns			
Branch mispredict	5	ns			
L2 cache reference	7	ns			14x L1 cache
Mutex lock/unlock	25	ns			
Main memory reference	100	ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000	ns	3	us	
Send 1K bytes over 1 Gbps network	10,000	ns	10	us	
Read 4K randomly from SSD*	150,000	ns	150	us	~1GB/sec SSD
Read 1 MB sequentially from memory	250,000	ns	250	us	
Round trip within same datacenter	500,000	ns	500	us	
Read 1 MB sequentially from SSD*	1,000,000	ns	1,000	us	1 ms ~1GB/sec SSD, 4X memory
Disk seek	10,000,000	ns	10,000	us	10 ms 20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000	ns	20,000	us	20 ms 80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150,000	us	150 ms

### Notes

-----

1 ns = 10<sup>-9</sup> seconds  
 1 us = 10<sup>-6</sup> seconds = 1,000 ns  
 1 ms = 10<sup>-3</sup> seconds = 1,000 us = 1,000,000 ns

### Credit

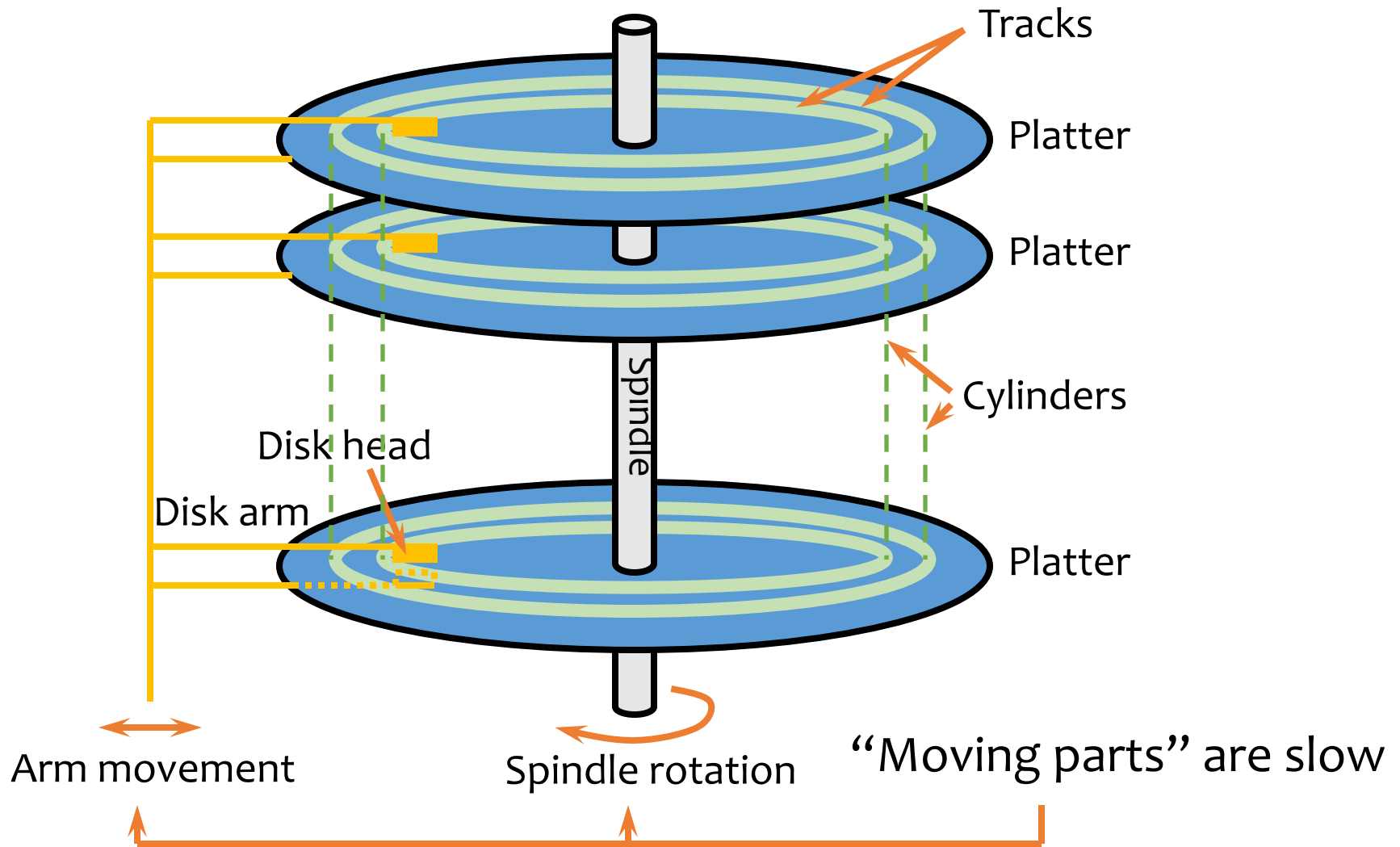
-----

By Jeff Dean: <http://research.google.com/people/jeff/>  
 Originally by Peter Norvig: <http://norvig.com/21-days.html#answers>

# A typical hard drive

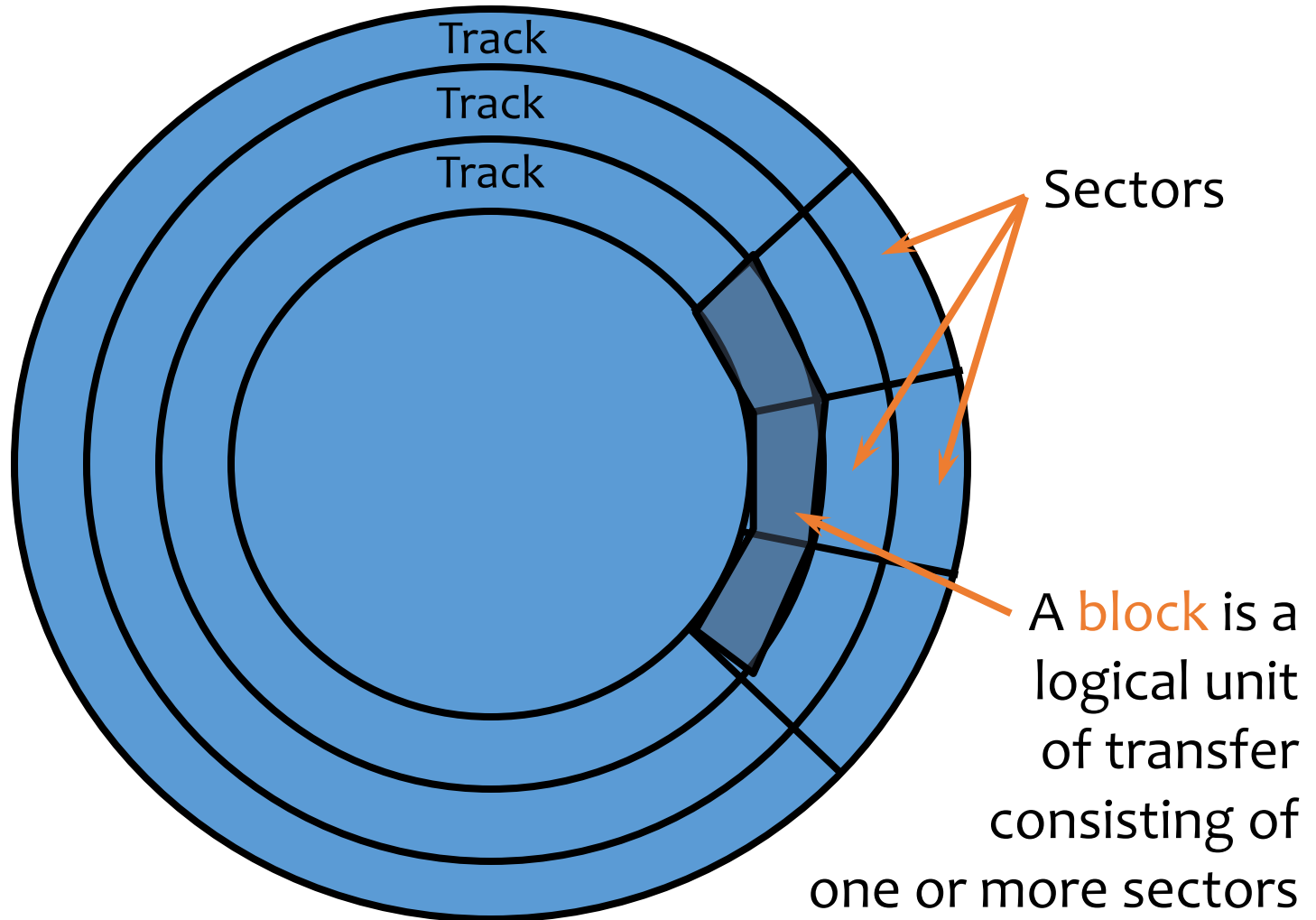


# A typical hard drive



# Top view

“Zoning”: more sectors/data on outer tracks





# Disk access time

Sum of:

- **Seek time**: time for disk heads to move to the correct cylinder
- **Rotational delay**: time for the desired block to rotate under the disk head
- **Transfer time**: time to read/write data in the block  
(= time for disk to rotate over the block)

# Random disk access

Seek time + rotational delay + transfer time

- Average seek time
  - Time to skip one half of the cylinders?
  - Not quite; time to skip a third of them
  - “Typical” value: 5 ms
- Average rotational delay
  - Time for a half rotation (a function of RPM)
  - “Typical” value: 4.2 ms (7200 RPM)

# Sequential disk access

Seek time + rotational delay + transfer time

- Seek time
  - 0 (assuming data is on the same track)
- Rotational delay
  - 0 (assuming data is in the next block on the track)
- Easily an order of magnitude faster than random disk access!

# What about SSD (solid-state drives)?



# What about SSD (solid-state drives)?

- No mechanical parts
- Mostly flash-based nowadays
- 1-2 orders of magnitude faster random access than hard drives (under 0.1ms vs. several ms)
  - But still much slower than memory ( $\sim 0.1\mu s$ )
- Little difference between random vs. sequential read performance
- Random writes still hurt
  - In-place update would require erasing the whole “erasure block” and rewriting it!

# Important consequences

- It's all about reducing I/O's!
- Cache blocks from stable storage in memory
  - DBMS maintains a memory **buffer pool** of blocks
  - Reads/writes operate on these memory blocks
  - Dirty (updated) memory blocks are “flushed” back to stable storage
- Sequential I/O is much faster than random I/O

# Performance tricks

- Disk layout strategy
  - Keep related things (what are they?) close together: same sector/block → same track → same cylinder → adjacent cylinder
- Prefetching
  - While processing the current block in memory, fetch the next block from disk (overlap I/O with processing)
- Parallel I/O
  - More disk heads working at the same time
- Disk scheduling algorithm
  - Example: “elevator” algorithm
- Track buffer
  - Read/write one entire track at a time

# Record layout

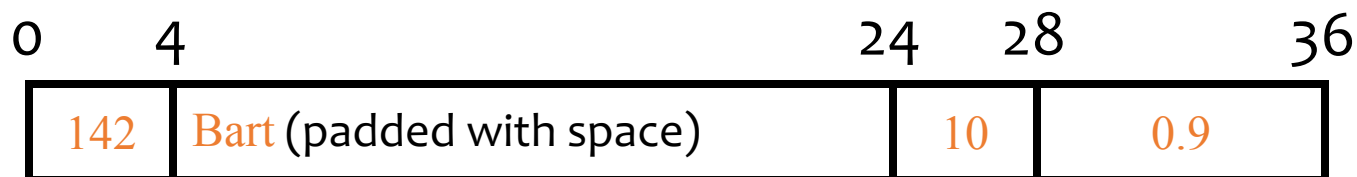
Record = row in a table

- Variable-format records
  - Rare in DBMS—table schema dictates the format
  - Relevant for semi-structured data such as XML
- Focus on fixed-format records
  - With fixed-length fields only, or
  - With possible variable-length fields



# Fixed-length fields

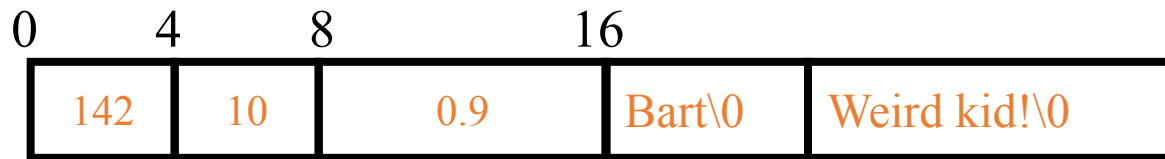
- All field lengths and offsets are constant
  - Computed from schema, stored in the system catalog
- Example: CREATE TABLE User(uid INT, name CHAR(20), age INT, pop FLOAT);



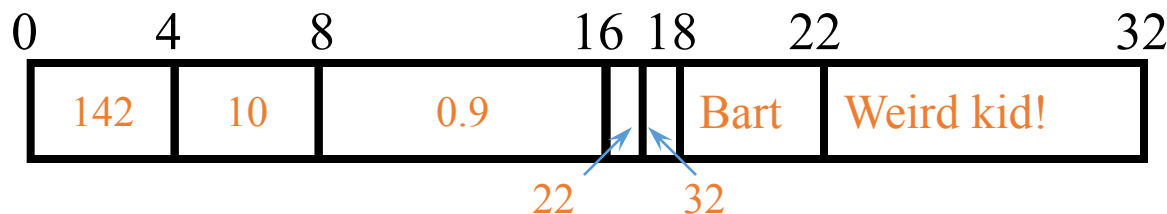
- Watch out for alignment
  - May need to pad; reorder columns if that helps
- What about NULL?
  - Add a bitmap at the beginning of the record

# Variable-length records

- Example: `CREATE TABLE User(uid INT, name VARCHAR(20), age INT, pop FLOAT, comment VARCHAR(100));`
- Approach 1: use field delimiters ('\\0' okay?)



- Approach 2: use an offset array



- Put all variable-length fields at the end (why?)
- Update is messy if it changes the length of a field

# LOB fields

- Example: CREATE TABLE User(uid INT, name CHAR(20), age INT, pop FLOAT, picture **BLOB(32000)**);
- Student records get “de-clustered”
  - Bad because most queries do not involve picture
- Decomposition (automatically and internally done by DBMS without affecting the user)
  - (uid, name, age, pop)
  - (uid, picture)

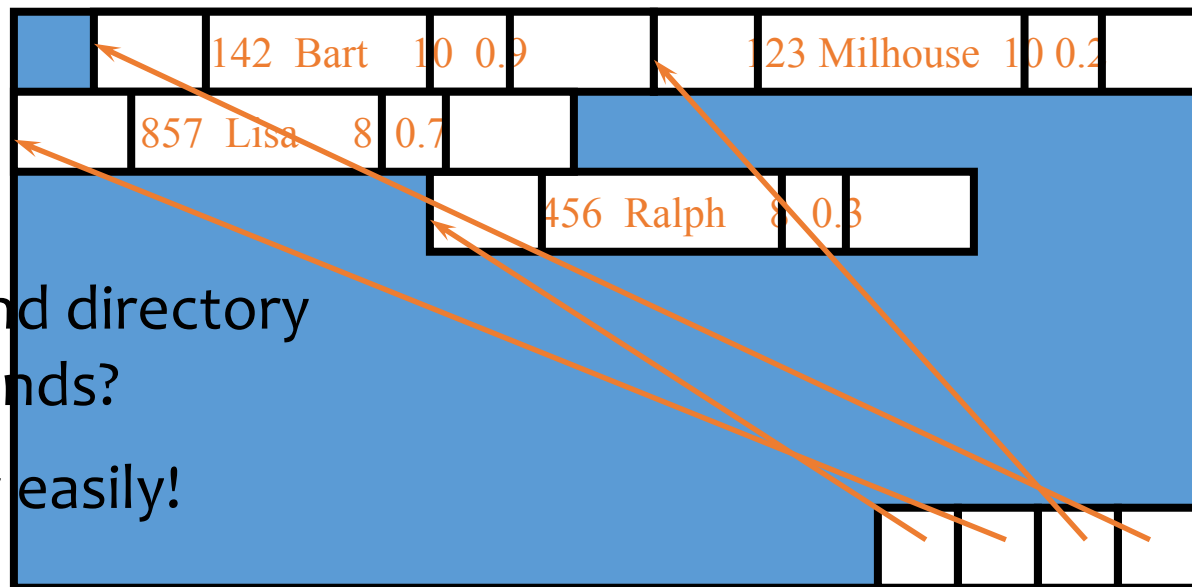
# Block layout

How do you organize records in a block?

- **NSM** (N-ary Storage Model)
  - Most commercial DBMS
- **PAX** (Partition Attributes Across)
  - Ailamaki et al., *VLDB* 2001

# NSM

- Store records from the beginning of each block
- Use a directory at the end of each block
  - To locate records and manage free space
  - Necessary for variable-length records



Why store data and directory  
at two different ends?

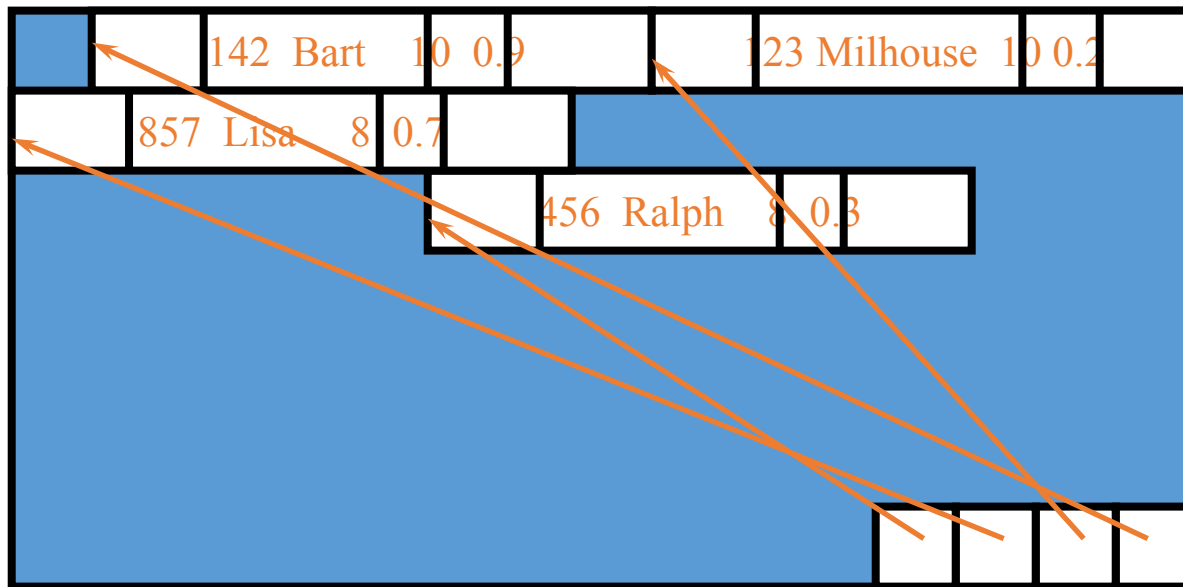
So both can grow easily!

# Options

- Reorganize after every update/delete to avoid fragmentation (gaps between records)
  - Need to rewrite half of the block on average
- A special case: What if records are fixed-length?
  - Option 1: reorganize after delete
    - Only need to move one record
    - Need a pointer to the beginning of free space
  - Option 2: do not reorganize after update
    - Need a bitmap indicating which slots are in use

# Cache behavior of NSM

- Query: `SELECT uid FROM User WHERE pop > 0.8;`
- Assumptions: no index, and cache line size < record size
- Lots of cache misses
  - uid and pop are not close enough by memory standards

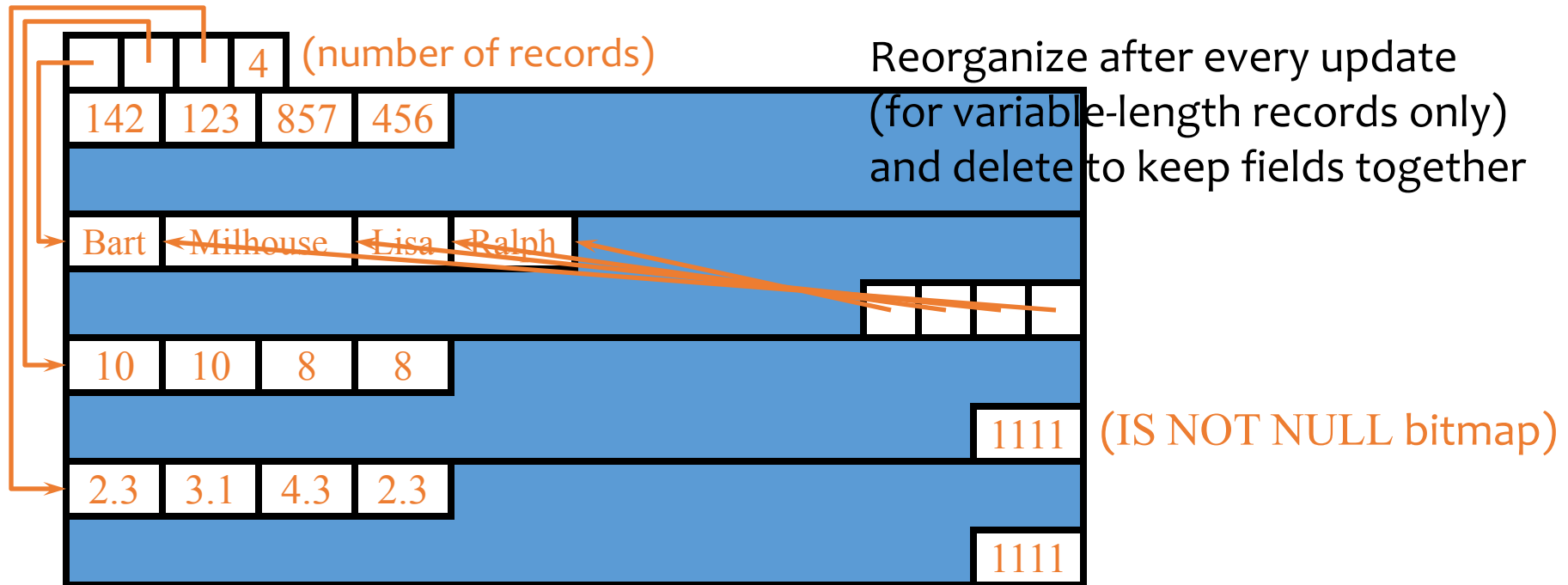


142 Bart 10
0.9 123 Milhouse
10 0.2 857 Lisa
8 0.7
456 Ralph 8
0.3

Cache

# PAX

- Most queries only access a few columns
- Cluster values of the same columns in each block
  - When a particular column of a row is brought into the cache, the same column of the next row is brought in together





# Beyond block layout: column stores

- The other extreme: store tables by columns instead of rows
- Advantages (and disadvantages) of PAX are magnified
  - Not only better cache performance, but also fewer I/O's for queries involving many rows but few columns
  - Aggressive compression to further reduce I/O's
- More disruptive changes to the DBMS architecture are required than PAX
  - Not only storage, but also query execution and optimization

# Example: Apache Parquet



- A table is horizontally partitioned into **row groups** (~512MB-1GB/row group); each group is stored consecutively
  - On a “block” of HDFS (Hadoop Distributed File System)
- A row group is vertically divided into **column chunks**, one per column
- Each column chunk is stored in **pages** (~8KB/page); each page can be compressed/encoded independently

👉 Not designed for in-place updates though!

# Summary

- Storage hierarchy
  - Why I/O's dominate the cost of database operations
- Disk
  - Steps in completing a disk access
  - Sequential versus random accesses
- Record layout
  - Handling variable-length fields
  - Handling NULL
  - Handling modifications
- Block layout
  - NSM: the traditional layout
  - PAX: a layout that tries to improve cache performance
- Column stores: NSM transposed, beyond blocks