

Physical Data Organization

Introduction to Databases

CompSci 316 Spring 2019

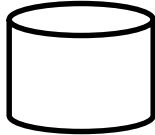


DUKE
COMPUTER SCIENCE

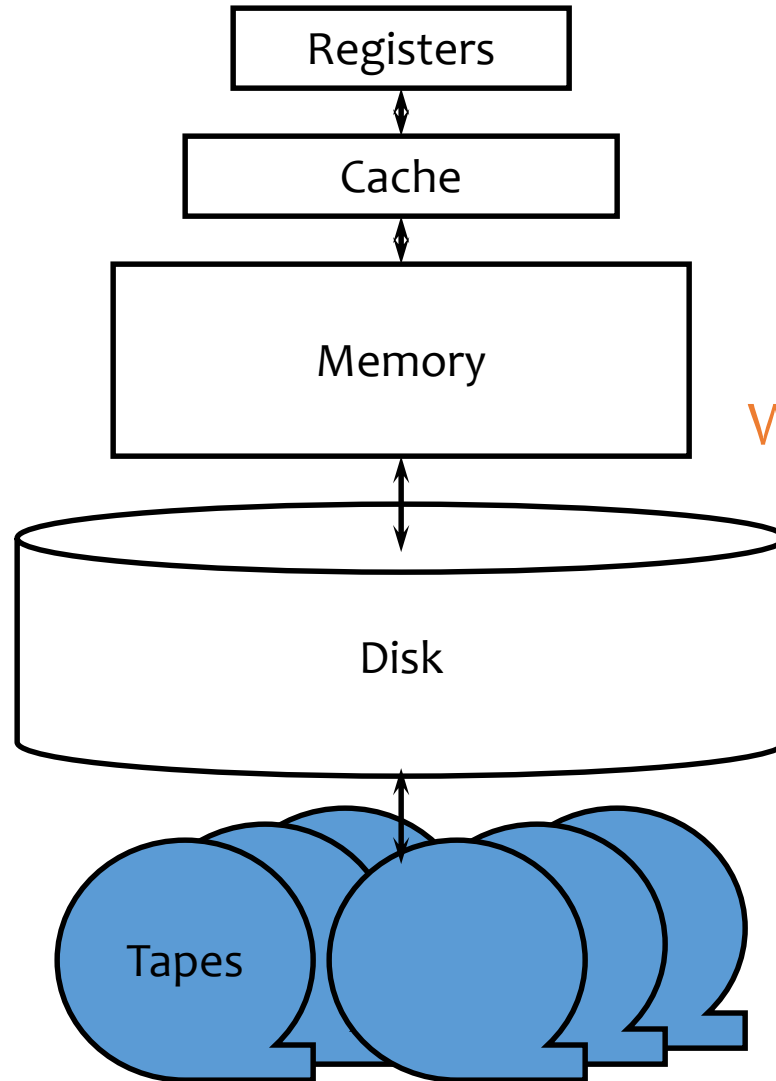
Announcements (Thu., Mar. 21)

- Homework #3 due on 03/27 – next Wednesday
- Project milestone #2 due next Friday 03/29 (extended by 3 days)
- Weekly progress update from all members of a group due from next week (Piazza post will follow)

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 = 10⁻⁹ seconds
 1 us = 10⁻⁶ seconds = 1,000 ns
 1 ms = 10⁻³ seconds = 1,000 us = 1,000,000 ns

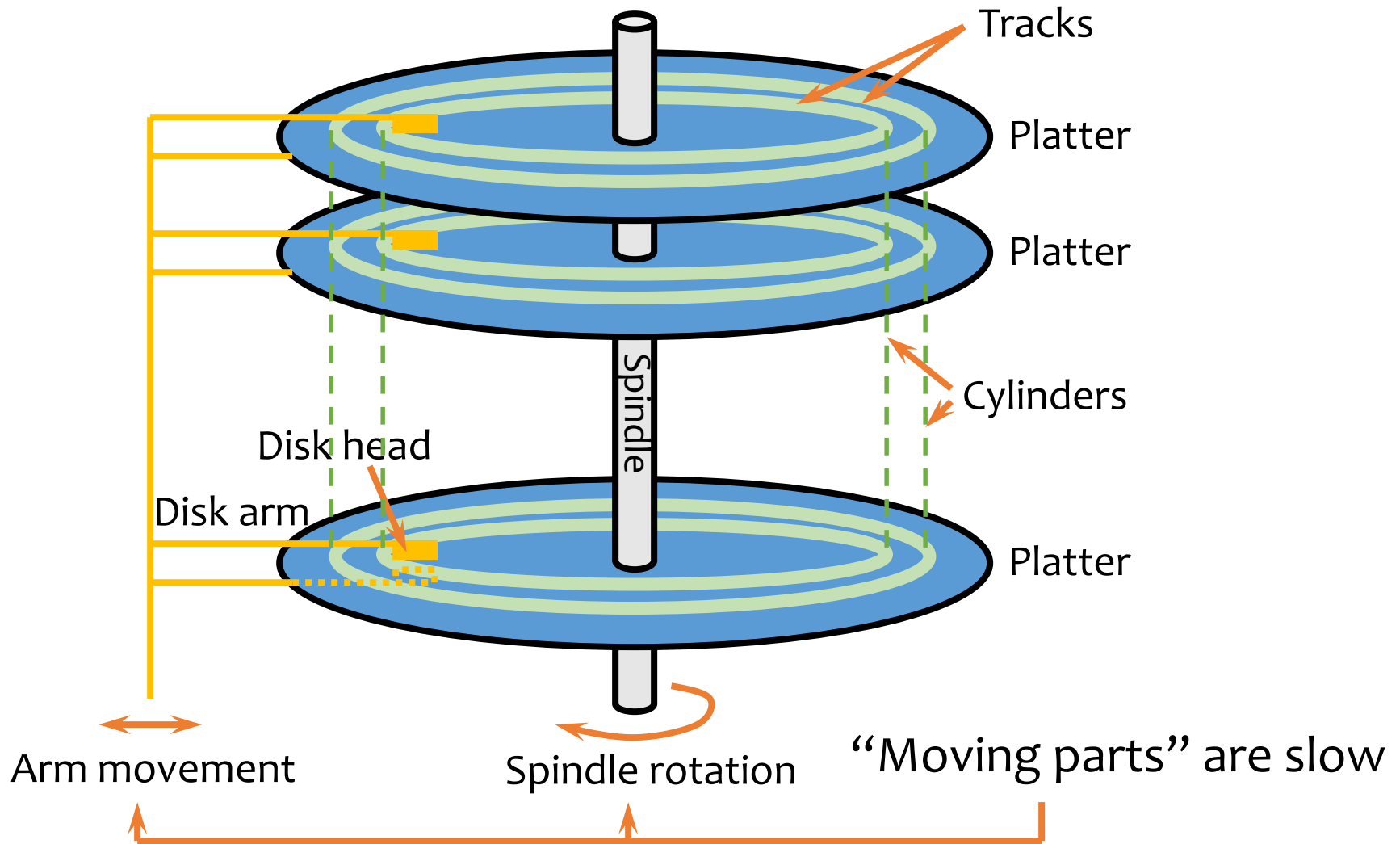
Credit

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

A typical hard drive

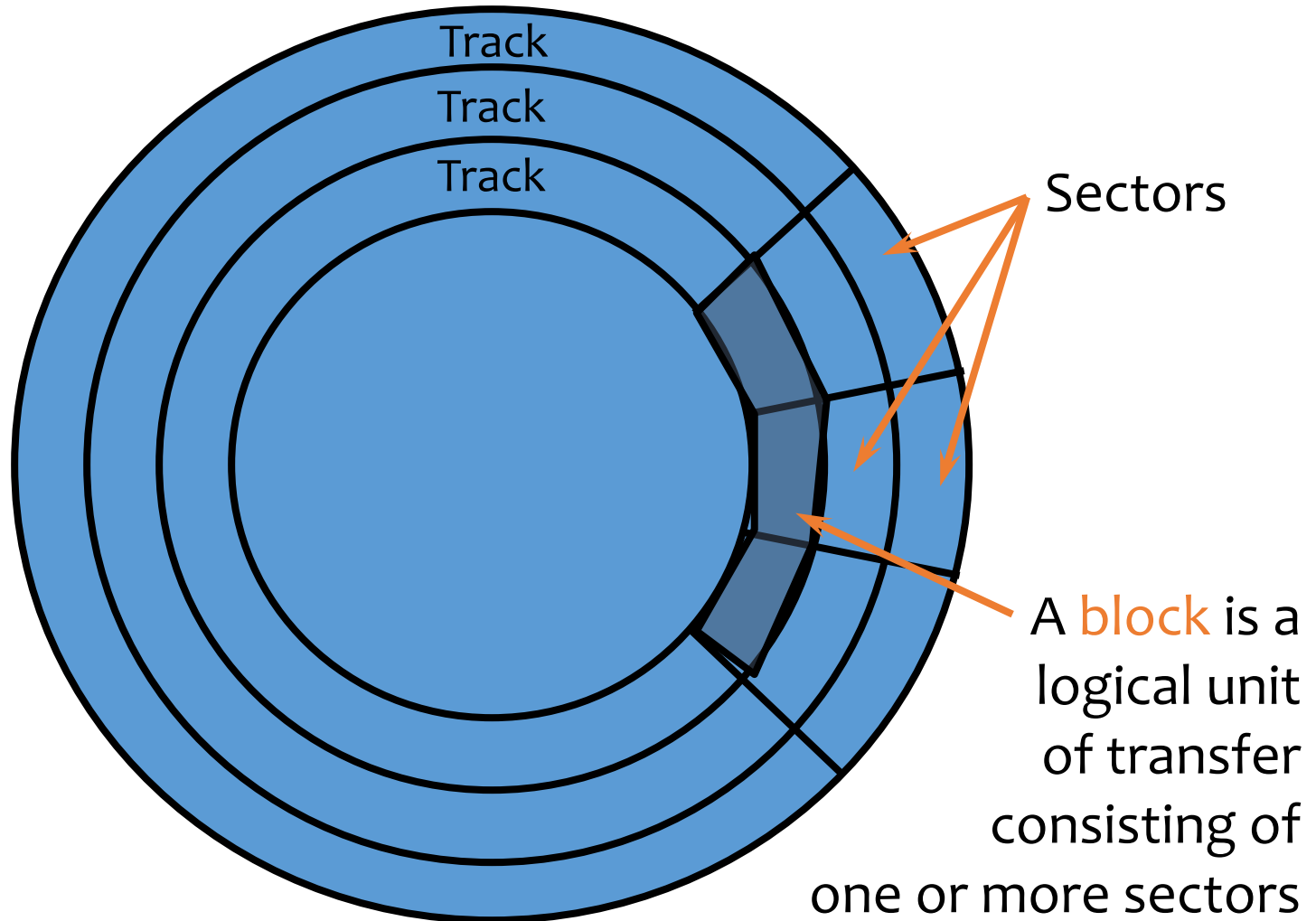


A typical hard drive



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; should be 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)?

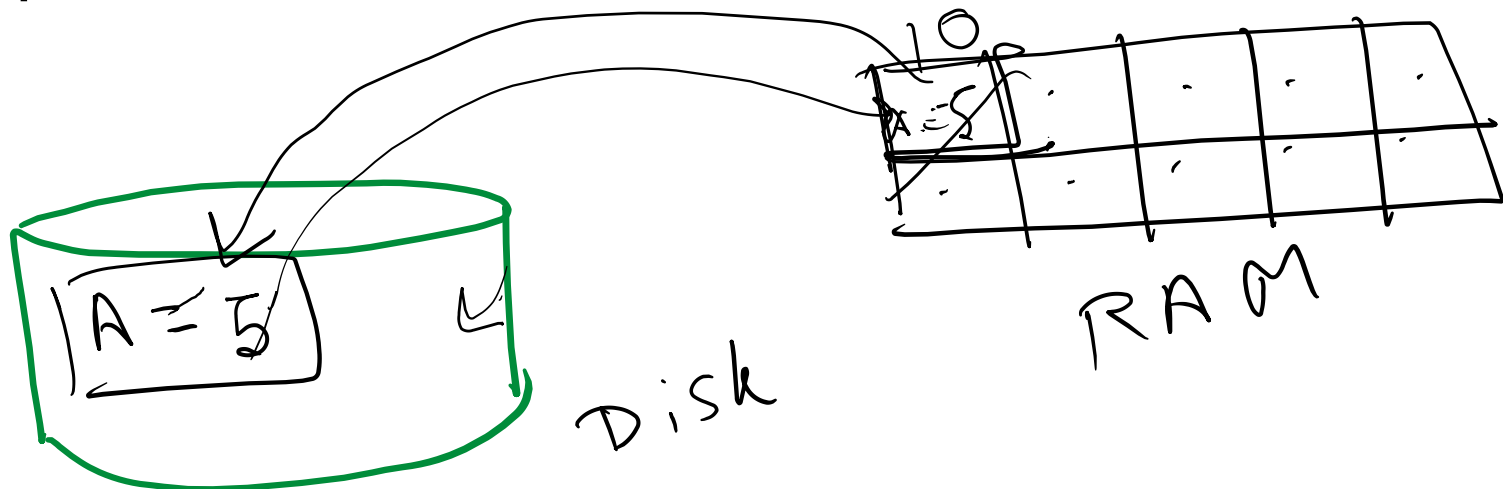


What about SSD (solid-state drives)?

- 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!

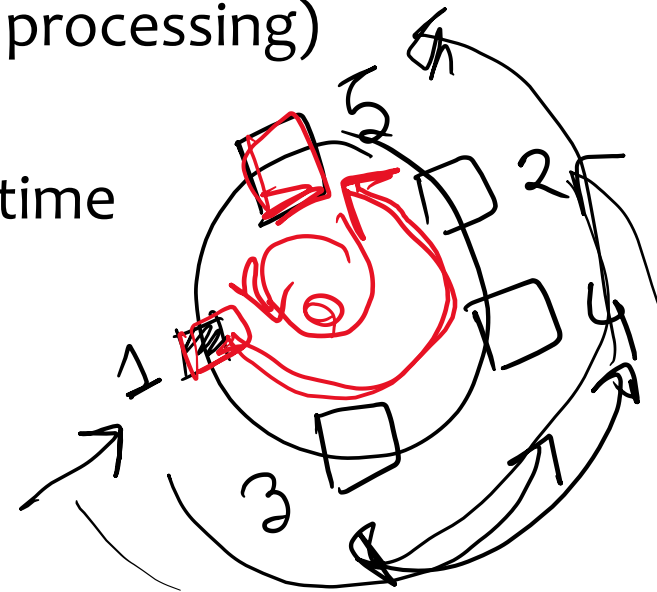
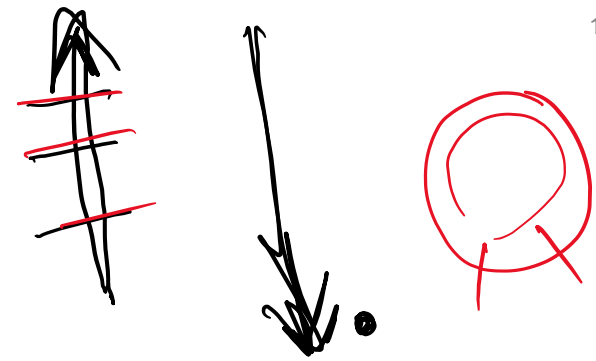
Important consequences

- 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



Hand-drawn diagram of a User table. The table has four columns: uid, name, age, and pop. The first row contains the values 4, 20, 9, and 8. A red arrow points down from the table.

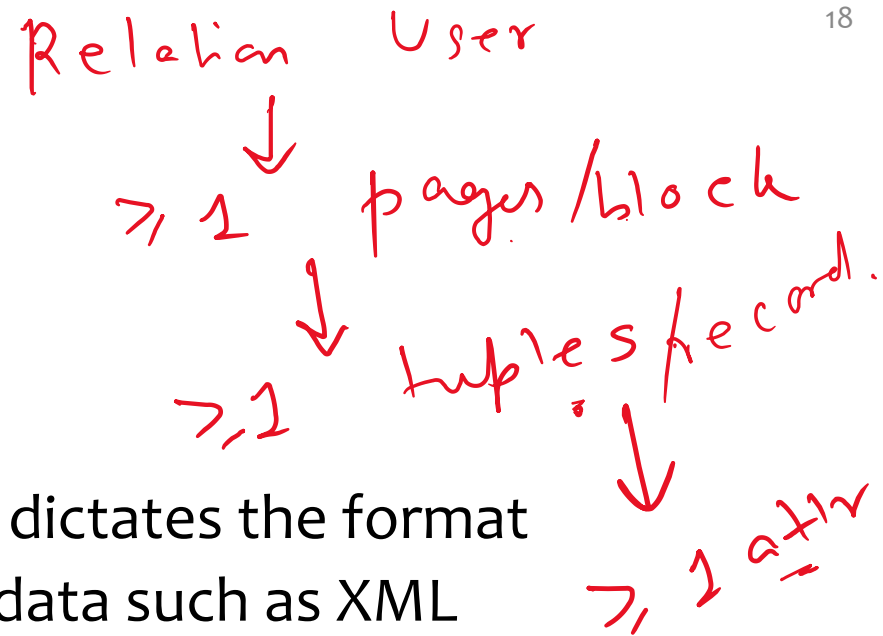
uid	name	age	pop
4	20	9	8

A hand-drawn diagram of a 1D array with 8 slots. The first slot is labeled 'Bert'. The second slot is empty. The third slot contains a horizontal line. The fourth slot is labeled '20'. The fifth slot is labeled '4'. The sixth slot is labeled '8'.

Record layout

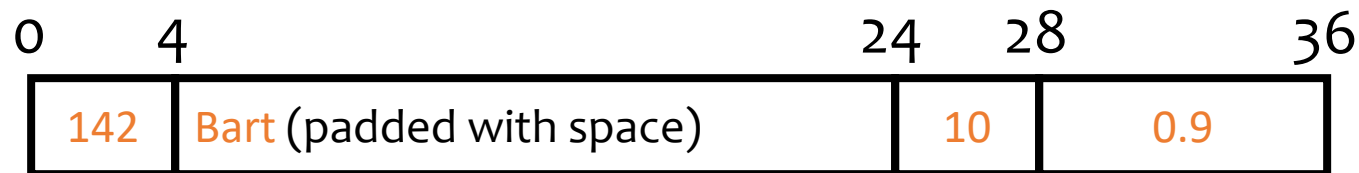
Record = row in a table

- 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

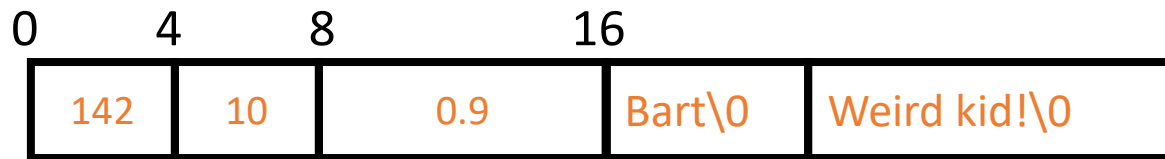
- All field lengths and offsets are constant
 - Computed from schema, stored in the system catalog
- Example: `CREATE TABLE User(uid INT, name CHAR(20), age INT, pop FLOAT);`



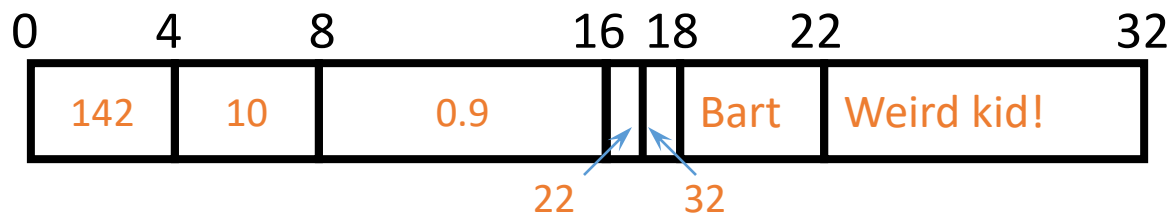
- Watch out for alignment
 - May need to pad; reorder columns if that helps
- What about NULL?
 - Add a bitmap at the beginning of the record

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?

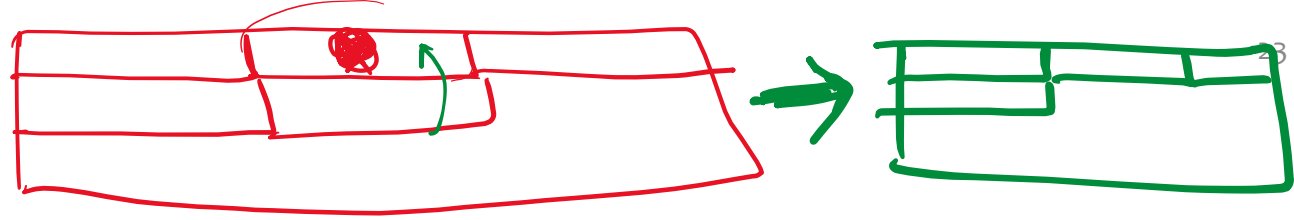
- **NSM** (N-ary Storage Model)
 - Most commercial DBMS
- **PAX** (Partition Attributes Across)
 - Ailamaki et al., VLDB 2001

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

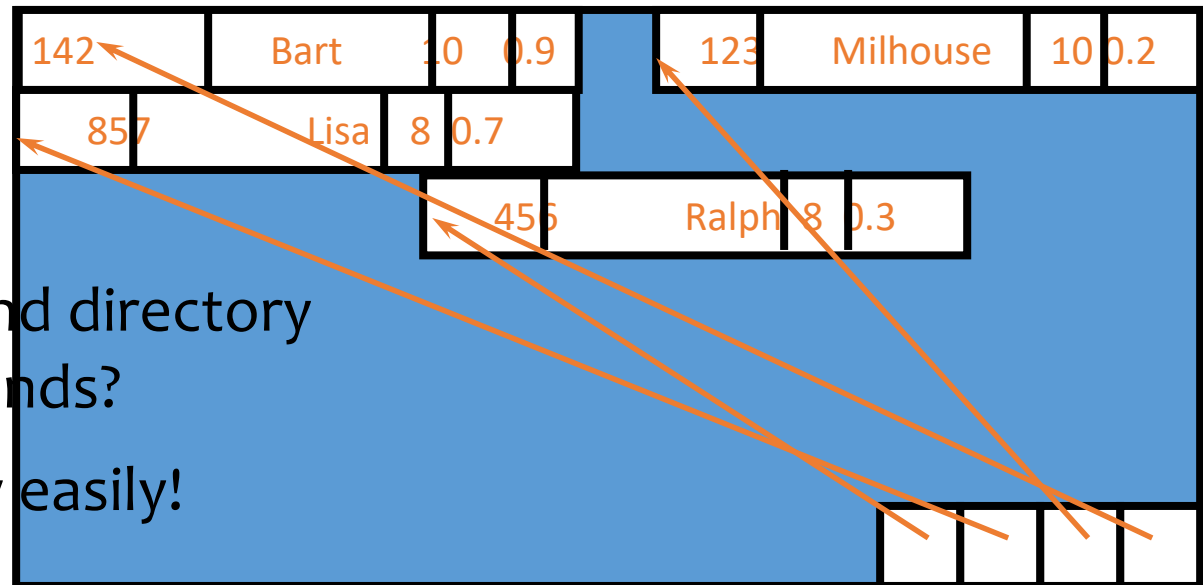
Row major order →
1 - 2 - 3 - 4 - 5 - 6 - ... - 15

Col-major order
1 - 5 - 9 - 13 - 2 - 6 - 10 - 14 - ... - 16

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?

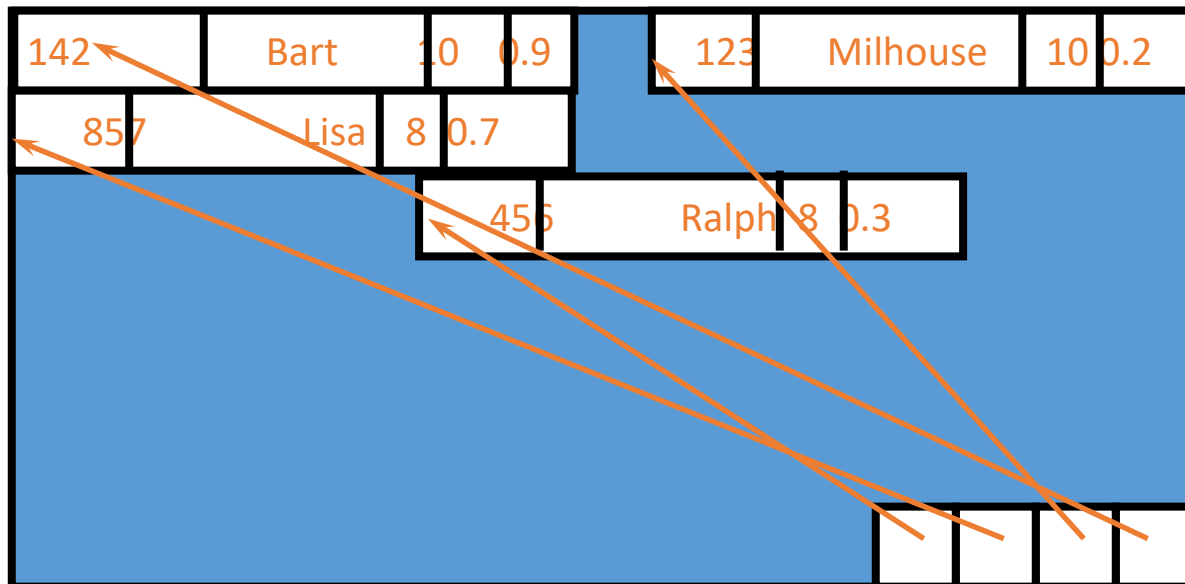
So both can grow easily!

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

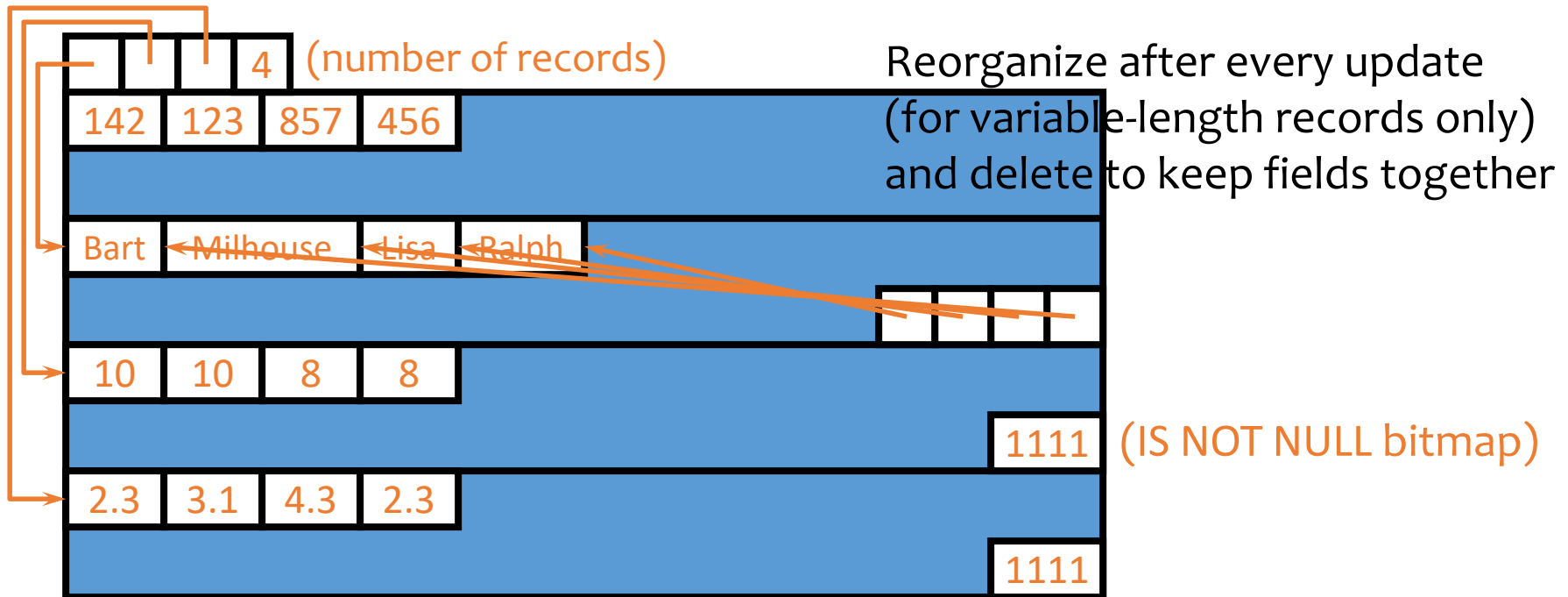


142 Bart 10
0.9 123 Milhouse
10 0.2 857 Lisa
8 0.7
456 Ralph 8
0.3

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
 - Examples: MonetDB, Vertica (earlier, C-store), SAP/Sybase IQ, Google Bigtable (with column groups)

Row store



+ update

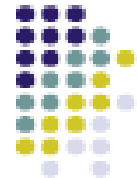
+ select

Col store



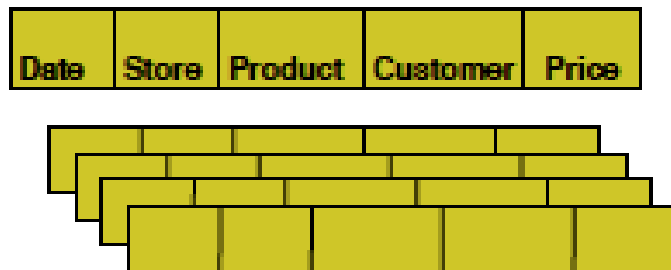
+ select age

Re-use permitted when acknowledging the original © Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2006)



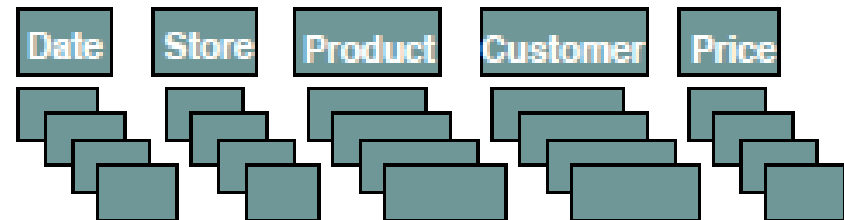
What is a column-store?

row-store



- + easy to add/modify a record
- might read in unnecessary data

column-store



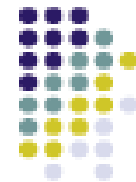
- + only need to read in relevant data
- tuple writes require multiple accesses

=> suitable for read-mostly, read-intensive, large data repositories

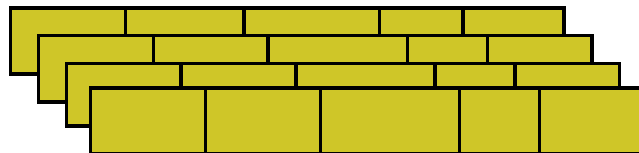
Ack: Slide from VLDB 2009 tutorial on Column store

Re-use permitted when acknowledging the original © Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2000)

Telco example explained (1/3): *read efficiency*



row store



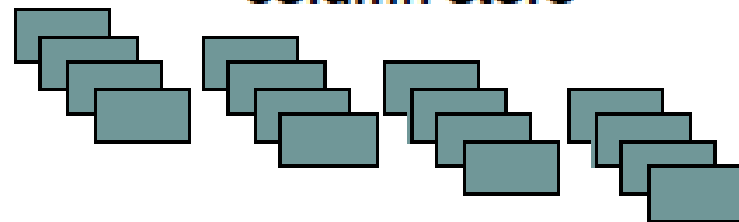
read pages containing entire rows

one row = 212 columns!

is this typical? (it depends)

What about vertical partitioning?
(it does not work with ad-hoc
queries)

column store



read only columns needed

in this example: 7 columns

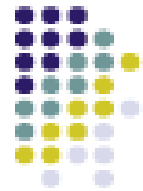
caveats:

- “select * ” not any faster
- clever disk prefetching
- clever tuple reconstruction

Ack: Slide from VLDB 2009 tutorial on Column store

Re-use permitted when acknowledging the original © Stergios Hatzopoulos, Daniel Abadi, Peter Boncz (2009)

Telco example explained (2/3): *compression efficiency*



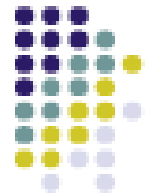
- 1 Columns compress better than rows
 - 1 Typical row-store compression ratio 1 : 3
 - 1 Column-store 1 : 10
- 1 Why?
 - 1 Rows contain values from different domains
=> more entropy, difficult to dense-pack
 - 1 Columns exhibit significantly less entropy
 - 1 Examples:

Male, Female, Female, Female, Male
1998, 1998, 1999, 1999, 1999, 2000
 - 1 Caveat: CPU cost (use lightweight compression)

Ack: Slide from VLDB 2009 tutorial on Column store

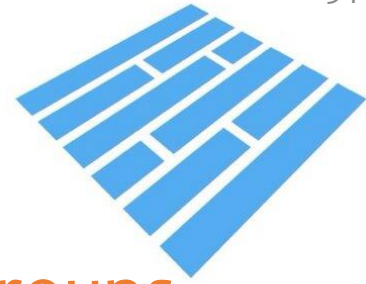
Re-use permitted when acknowledging the original © Sterros Harizopoulos, Daniel Abadi, Peter Boncz (2000)

Telco example explained (3/3): *sorting & indexing efficiency*



- 1 Compression and dense-packing free up space
 - 1 Use multiple overlapping column collections
 - 1 Sorted columns compress better
 - 1 Range queries are faster
 - 1 Use sparse clustered indexes

Ack: Slide from VLDB 2009 tutorial on Column store

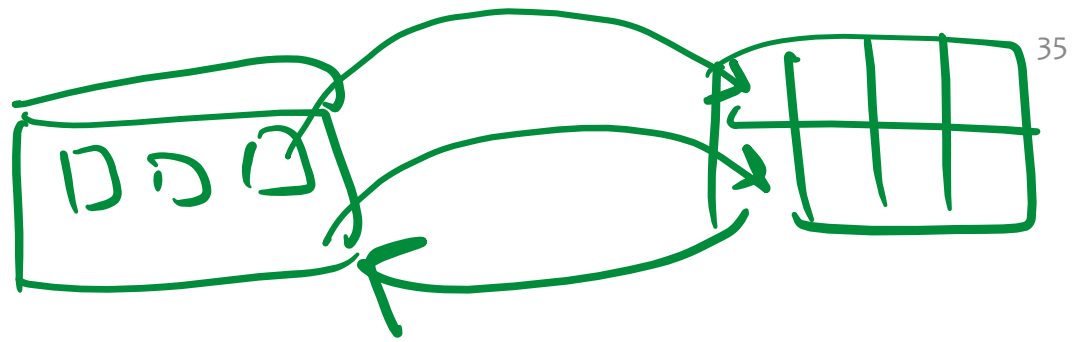


Example: Apache Parquet

- 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