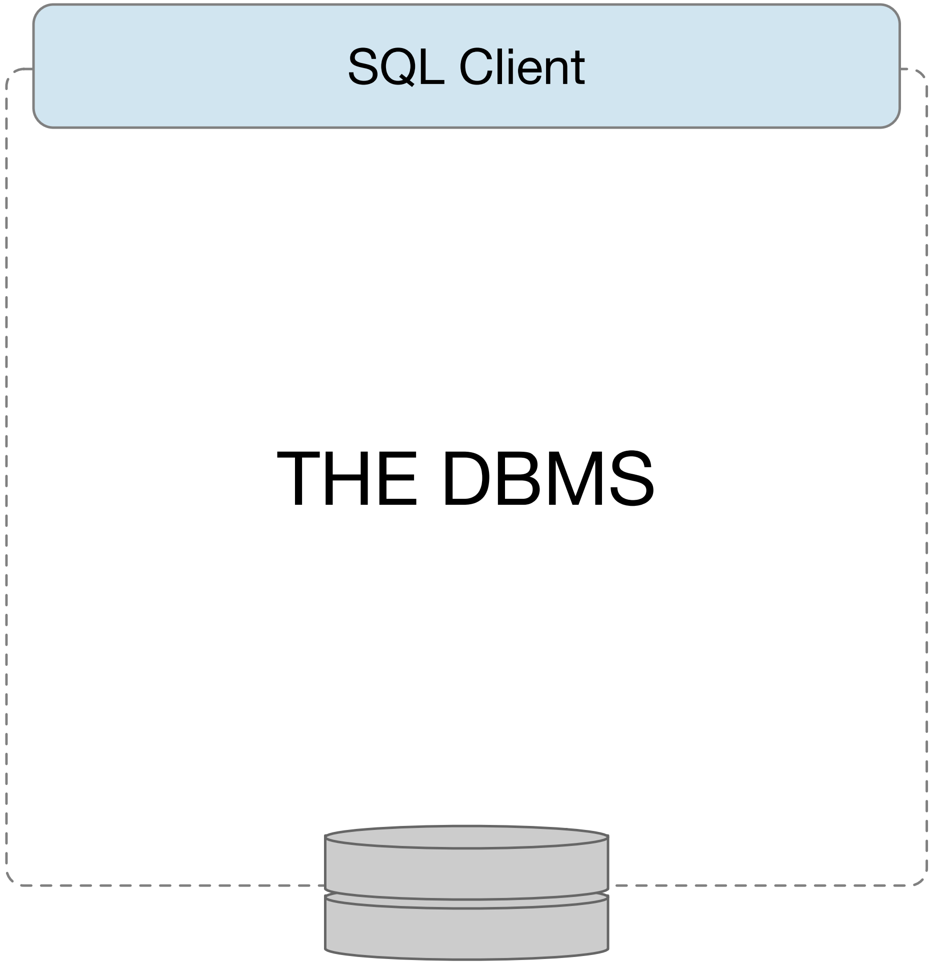
The insides of a DBMS

How is a SQL query executed?

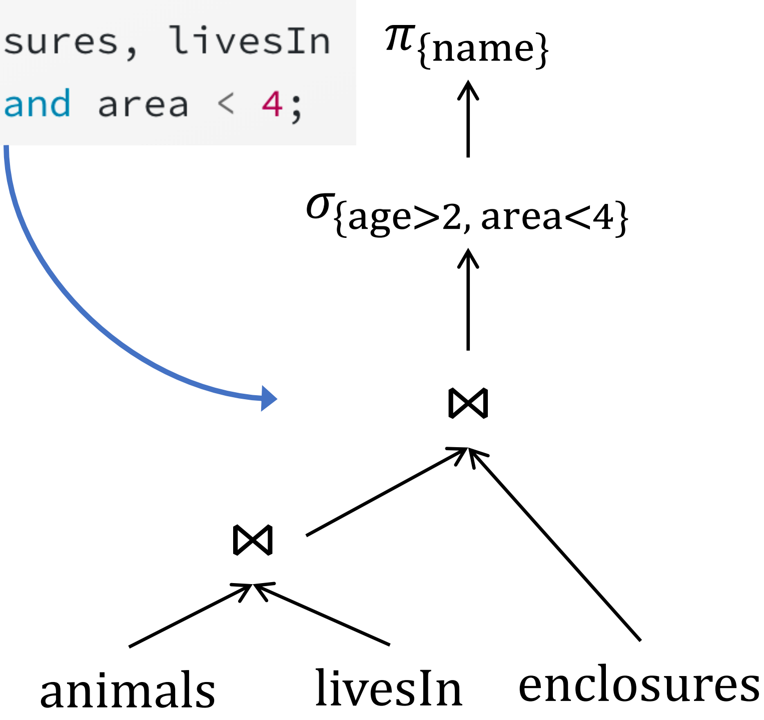


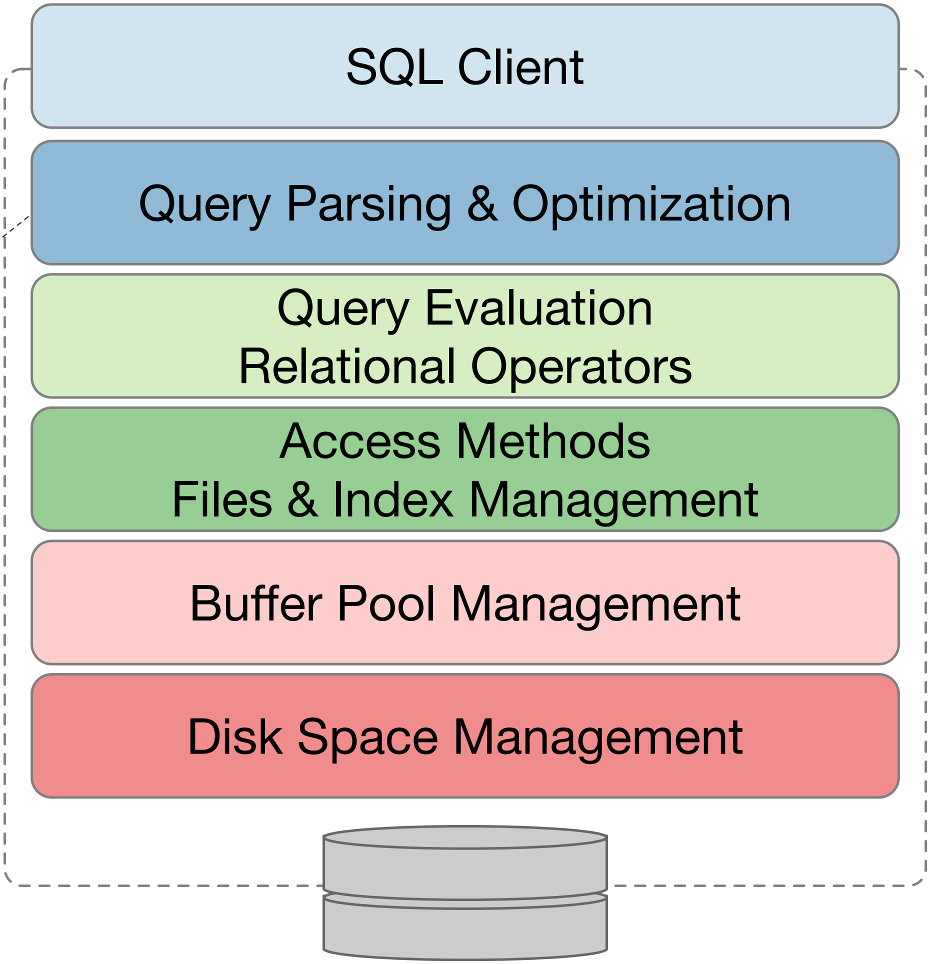
SQL Client

THE DBMS

Parse, Check, Rewrite, Optimize

```
select name from
animals, enclosures, livesIn
where age > 2 and area < 4;
```





SQL Client

Query Parsing & Optimization

Query Evaluation
Relational Operators

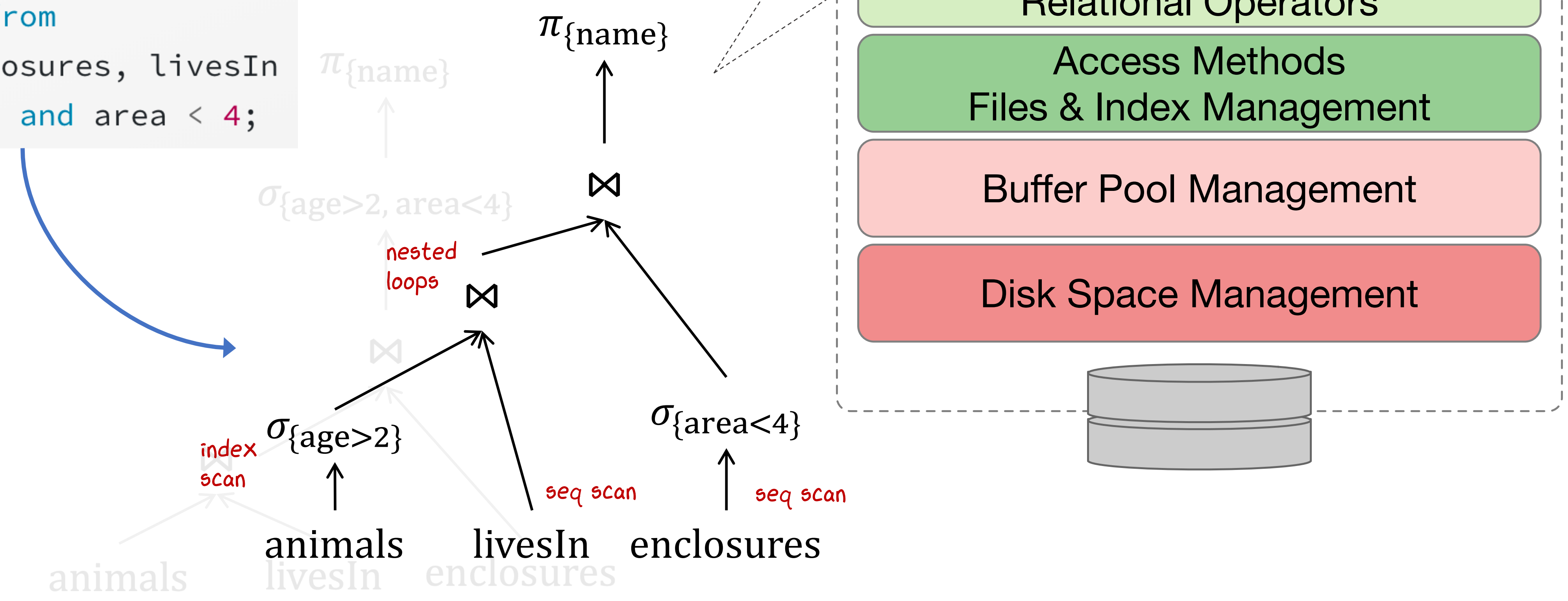
Access Methods
Files & Index Management

Buffer Pool Management

Disk Space Management

Parse, Check, Rewrite, Optimize

```sql
select name from
animals, enclosures, livesIn
where age > 2 and area < 4;
```



SQL Client

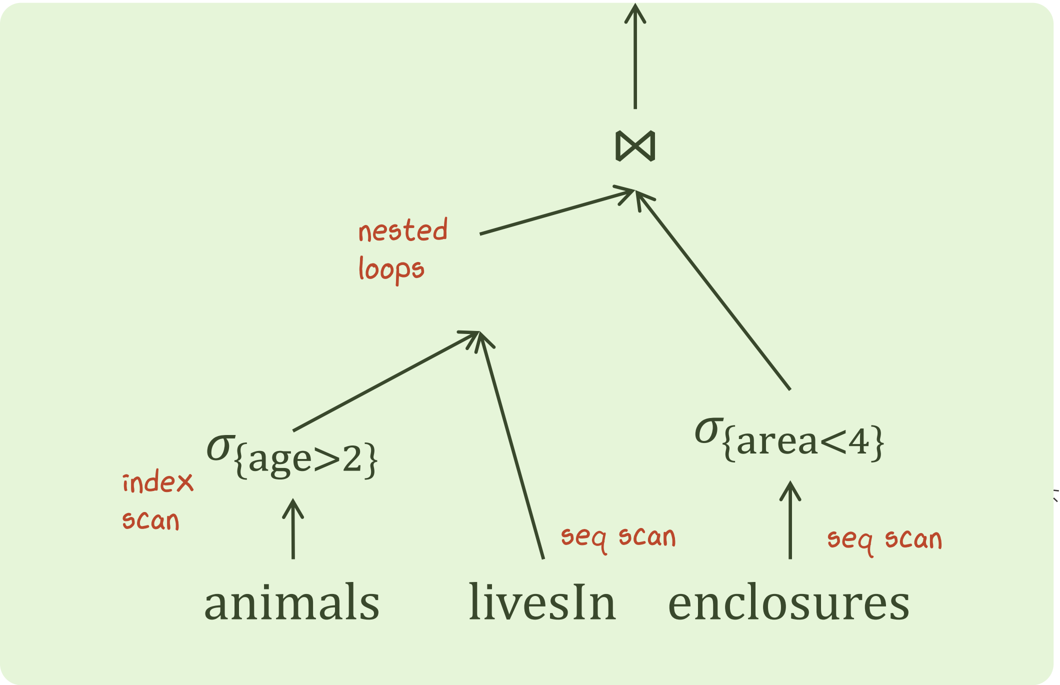
Query Parsing & Optimization

Query Evaluation
Relational Operators

Access Methods
Files & Index Management

Buffer Pool Management

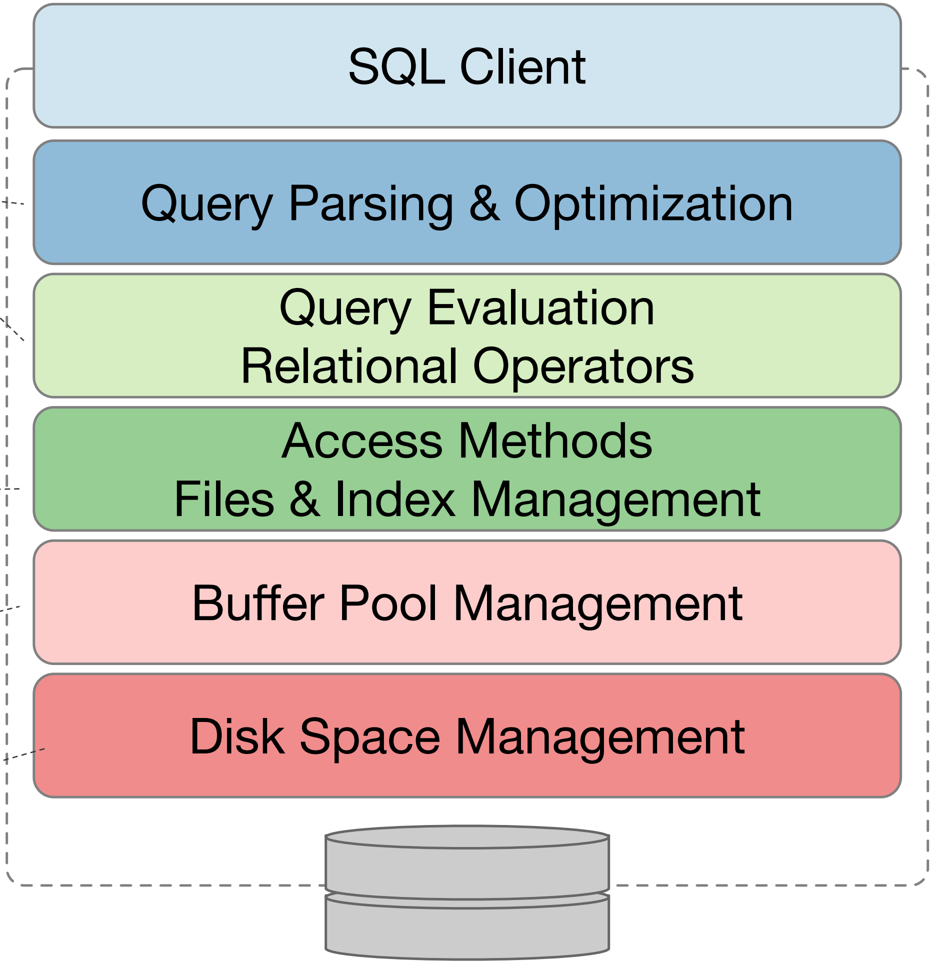
Disk Space Management



Organize tables and records as groups of pages in a *logical file*

Manage the transfer of pages into RAM to provide the *illusion* of operating in memory

(De)allocate, read, write *pages* on one or more storage device(s)



SQL Client

Query Parsing & Optimization

Query Evaluation
Relational Operators

Access Methods
Files & Index Management

Buffer Pool Management

Disk Space Management

The DBMS Architecture

Client-Server	• A SQL client for applications and a server DBMS
Layered	• *Manages complexity*: each layer abstracts and hides complexity from the layer above so it can focus on one thing!
Designed for disks	• Disks (HDDs) are slow and mechanical. • No byte-level addressing or pointer-dereferencing. Disk access is *block-level* or page-level. • An *API* to read page from disk to memory or write a page from memory to disk.

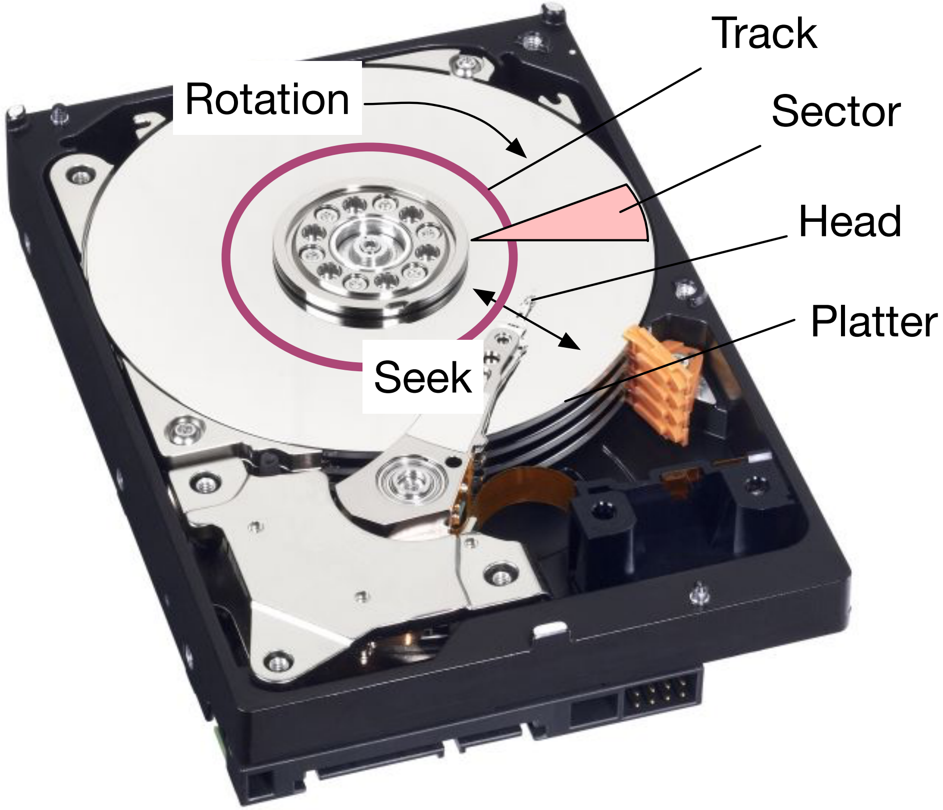
Disk Space Management

Disk Characteristics

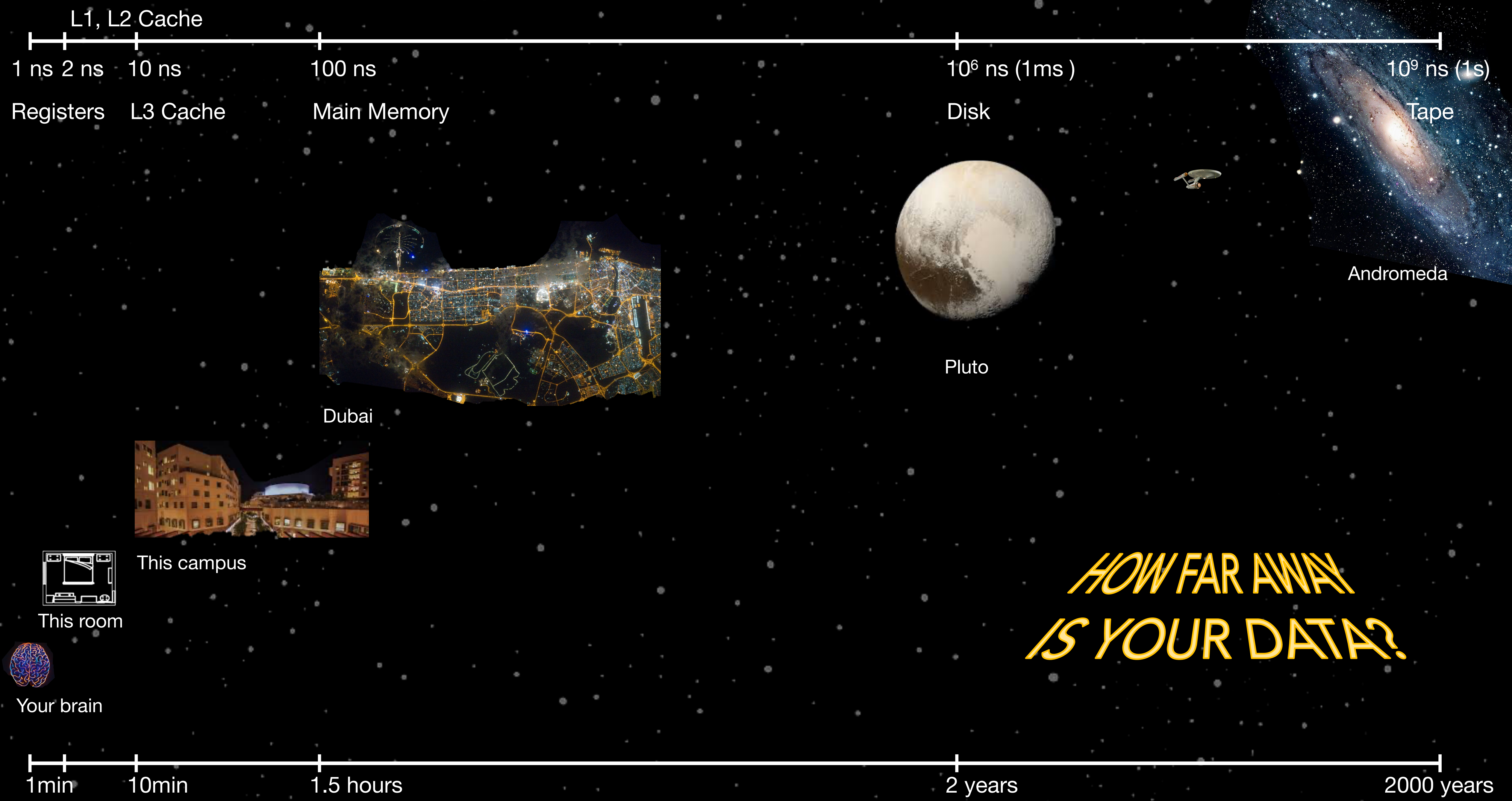
Time to retrieve a page depends on location:
seek (1-10ms) +
rotational delays (10ms).

Disk *bandwidth* is high: 100MB/s < page/1ms.

Arrange pages of a files *sequentially*! (next block on same track, same cylinder, adjacent cylinder).



HOW FAR AWAY IS YOUR DATA?



L1, L2 Cache

1 ns 2 ns 10 ns 100 ns 10^6 ns (1ms) 10^9 ns (1s)

Registers L3 Cache Main Memory Disk Tape

Your brain

This room

This campus

Dubai

Pluto

Andromeda

1min 10min 1.5 hours 2 years 2000 years

Jim Gray's Analogy

Disk & Storage Hierarchy Implications

Read and write large chunks of sequential bytes

- Reads and updates *occur at the block level* - a block has multiple records of many fields
- Typical Block/Page size (unit of transfer to and from storage) nowadays is 64-128 KB (postgres 8KB)

Sequential Access is fast

- "Next" disk block is fastest (no seek delays)
- *Spatial locality*: place things that are accessed together close to each other spatially

Amortize read and write costs

- Predict future access patterns & Prefetch blocks
- Cache popular blocks
- Buffer writes to sequential blocks

Disk Space Manager

API

- allocate/de-allocate a (sequence of) page(s)
- read/write a page

A single physical file can be stripped with pages across multiple devices each managed by a file system. The Disk Space manager provides the illusion of single big physical file with a unified getPage().

IMPLEMENTAION OPTIONS

Deal directly with the storage device

- Better control over where you place blocks and when you retrieve them
- What happens if the disk changes?

Run over the OS-provided File System

- Easier to implement & portable
- Allocate single large "contiguous" file; assume sequential access is fast. Most FS optimize disk layout for sequential access

Database Files

Main Questions

How are the contents of a DB file physically represented?	A database ***file*** is a collection of ***pages***, each containing a collection of ***records***.
What are different file organizations?	Unordered, Sorted, Indexed, …
How to choose the appropriate file organization?	Cost models and cost analysis

Layouts

Record Layouts

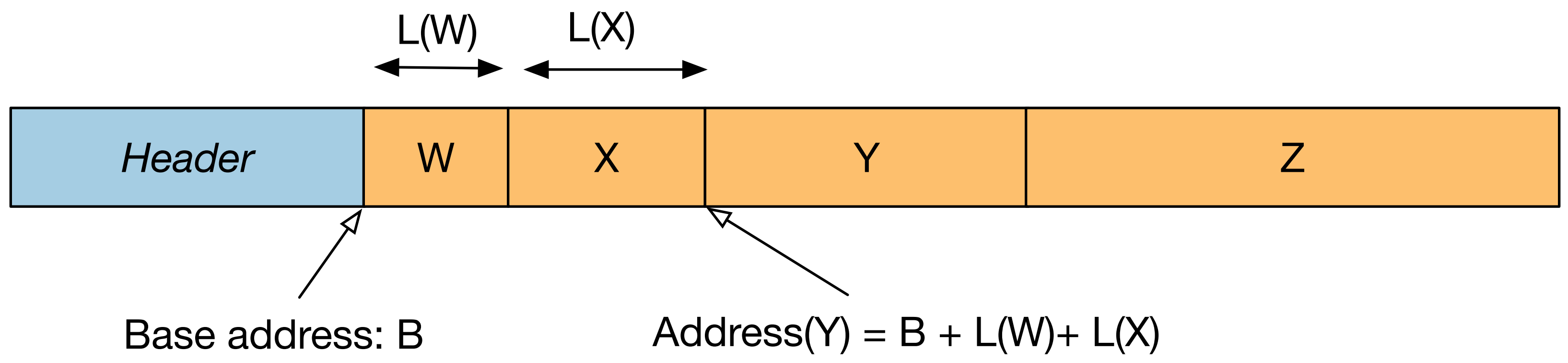
Record Layout

A sequence of bytes that are interpreted by the DBMS into attribute types and values.

DESIGN GOALS

1. Fast access to fields
2. Compact representation in both memory and disk
3. Handling both fixed and variable-length fields

To get the value of attribute Y:



Base address: B

Address(Y) = B + L(W)+ L(X)

Record Layout

Fixed Length Attributes

The *catalog* stores the schema; data type gives you length of each attribute.

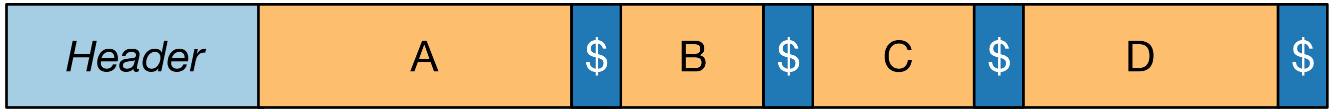
The order of fields is often the same order of the table definition

- Arithmetic is very fast!
- Byte representation on disk and in memory are identical
- Compact

Record Layout

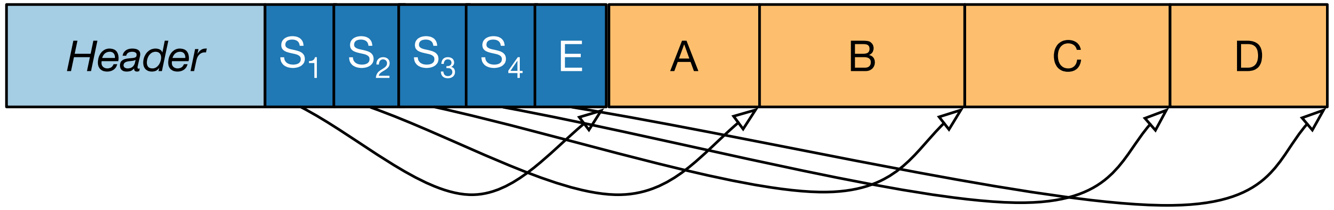
Variable Length Attributes

Option 1: Use Delimiters

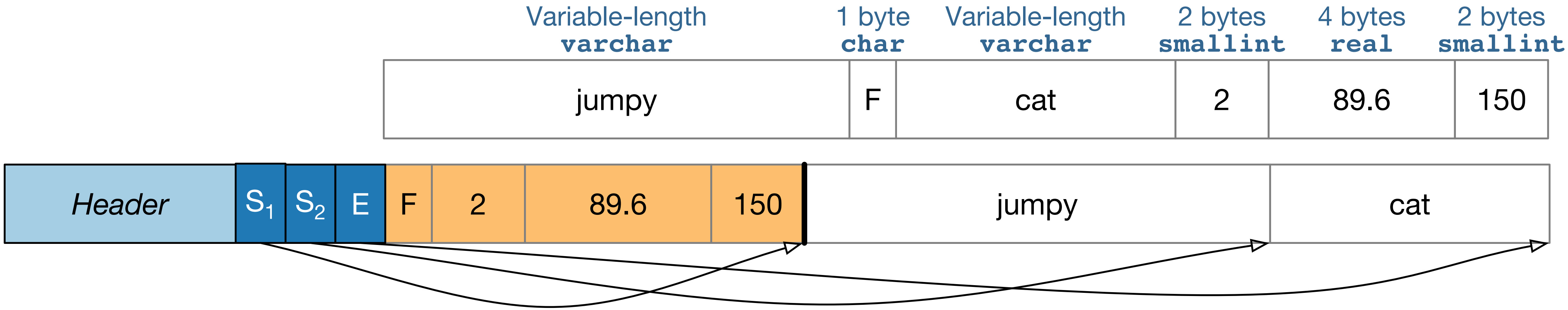


- Requires a full record scan to access each field
- What if the fields contain the delimiter?

Option 2: Use an array of field offsets



- Direct access to fields
- No need to escape delimiters



Record Layout

Fixed + Variable Length Attributes

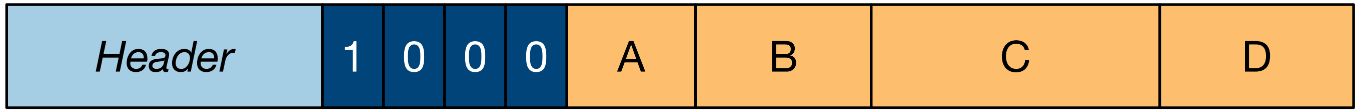
- Direct access to fixed-length fields
- Compact representation, only pointers for variable-length fields
- No need to escape delimiters

… But what about nulls?

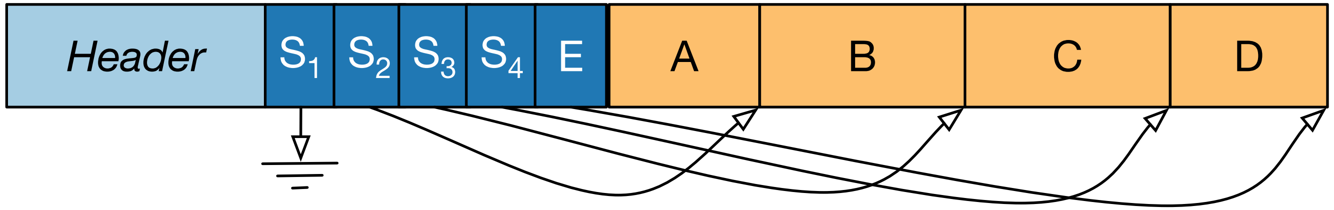
Record Layout

Handling Nulls

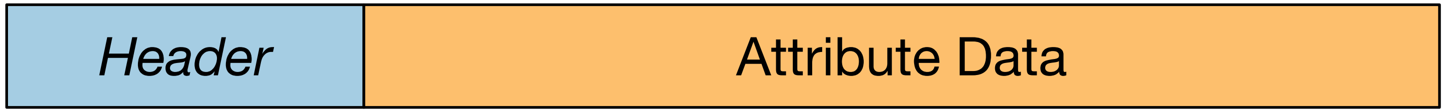
Option 1: Null bit map



Option 2: Pointers can handle this naturally



Record Layout



What goes in the header?

Header contains meta data:

- Does not contain schema
- Bit map for NULL values for fixed-length attributes
- Visibility information (revisit later in concurrency control)

What if we can't fit a record in a page?

A postgres page ~8KB

- Generally, a tuple cannot exceed the size of a single page.
- Use separate overflow storage page and store pointer to overflow page for field value
- Store pointers to external files (no durability or transaction guarantees)

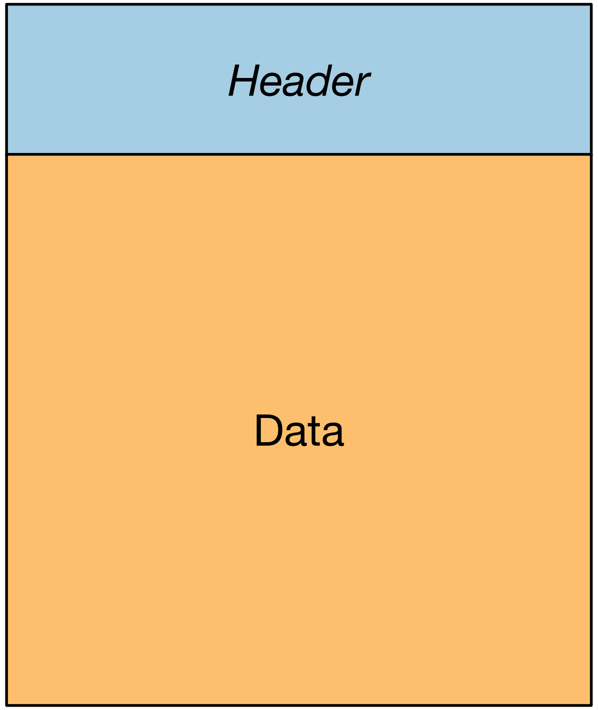
Page Layouts

Page Layout

A page has records of the same relation.
Each record is uniquely identified by a *rid*

DESIGN GOALS

1. Fast access to records
    - *By **rid** (page id, location in page)*
2. Compact representation in both memory and disk
3. Efficient handling of record deletes/inserts
4. Handles fragmentation



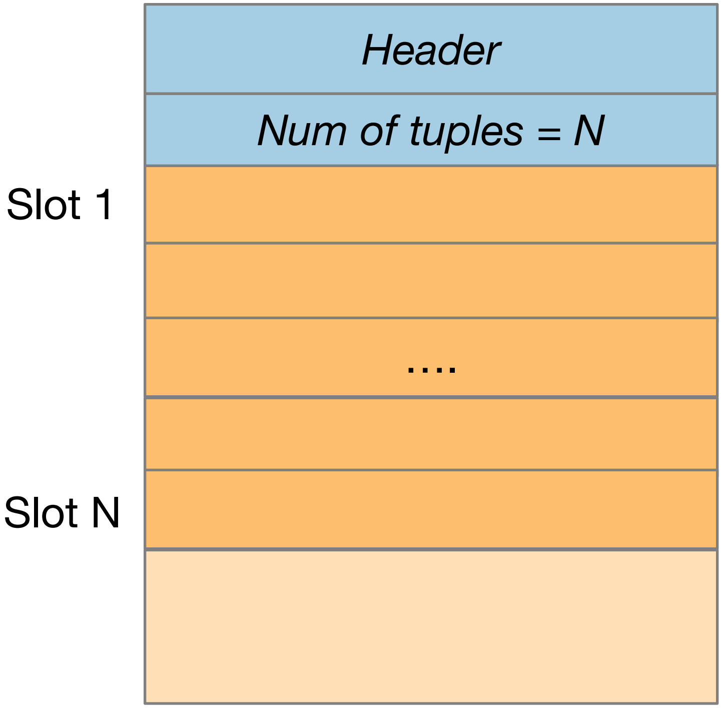
Page Layout

What goes in the header?

Header contains meta data:

- Number of records
- Free space
- Maybe a next/last pointer
- Bitmaps, Slot Table
- Page size
- Checksum
- DBMS version
- Visibility information
- Compression details

Option 1: Packed Fixed-Length Format



Pack records densely

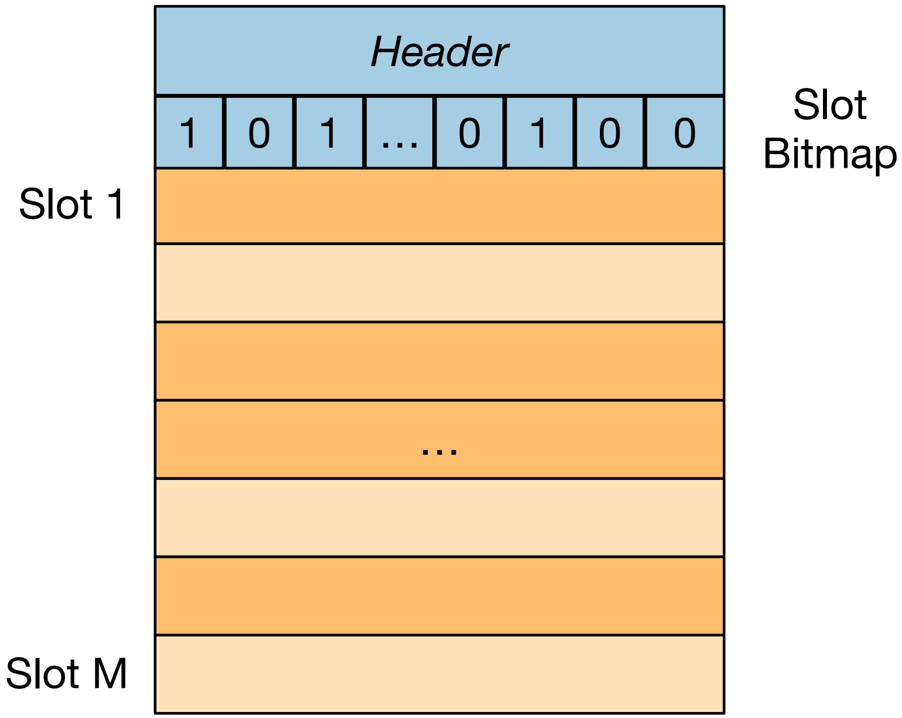
rid = (page id, "location in page") = (page id, slot #)

Insertions

- Just append

Deletions

- Re-arrange & update rids
- *What about other files (e.g. indexes) that may reference rids?*



Option 2: Unpacked Fixed-Length Format

Bitmap denotes "slots" with records

rid = (page id, slot #)

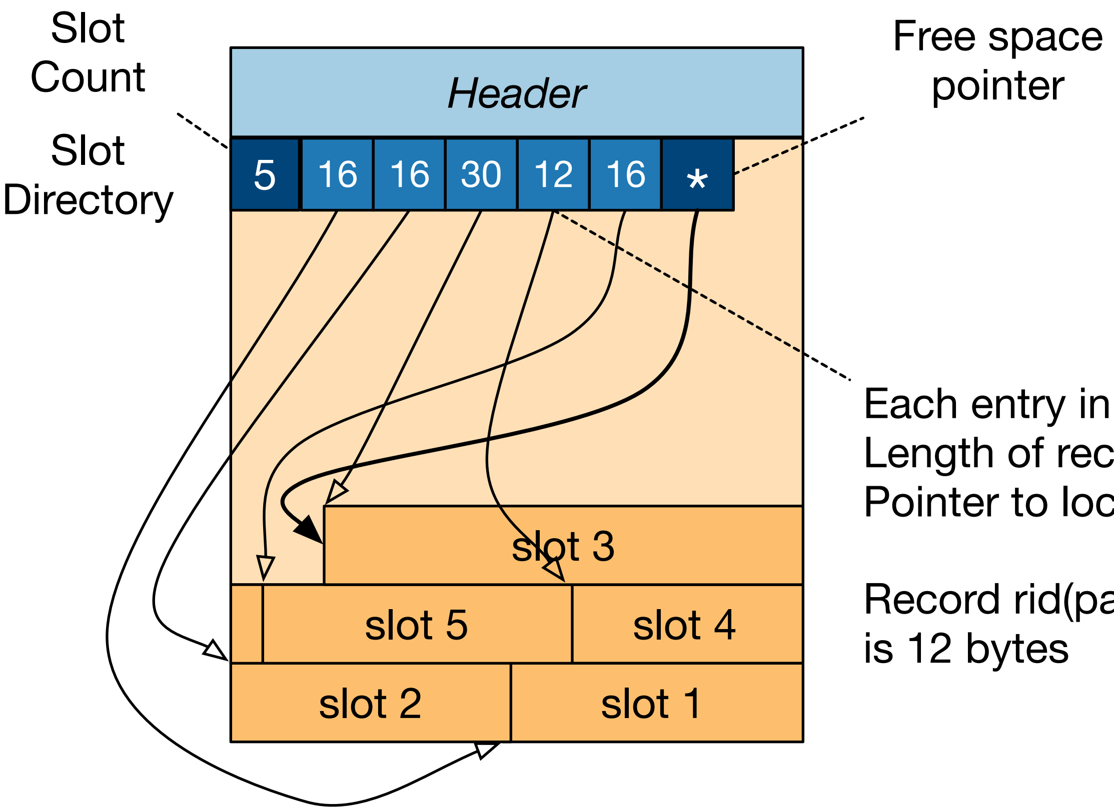
Insertions

- Find first empty slot

Deletions

- Clear bit

Option 3: Slotted Pages

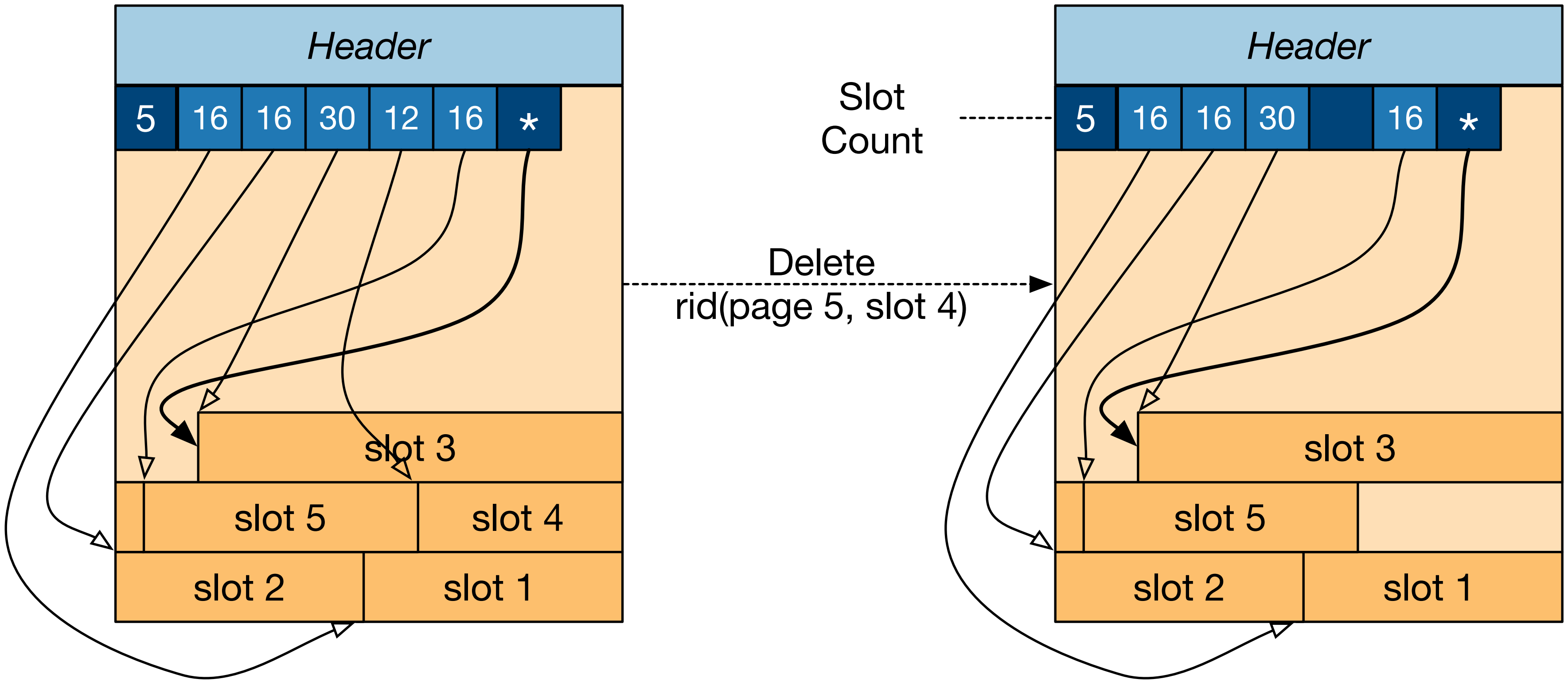


Each entry in slot array:
Length of record +
Pointer to location

Record rid(page 5, slot 4)
is 12 bytes

Each entry in the slot directory is a pointer to a record's beginning and its length.

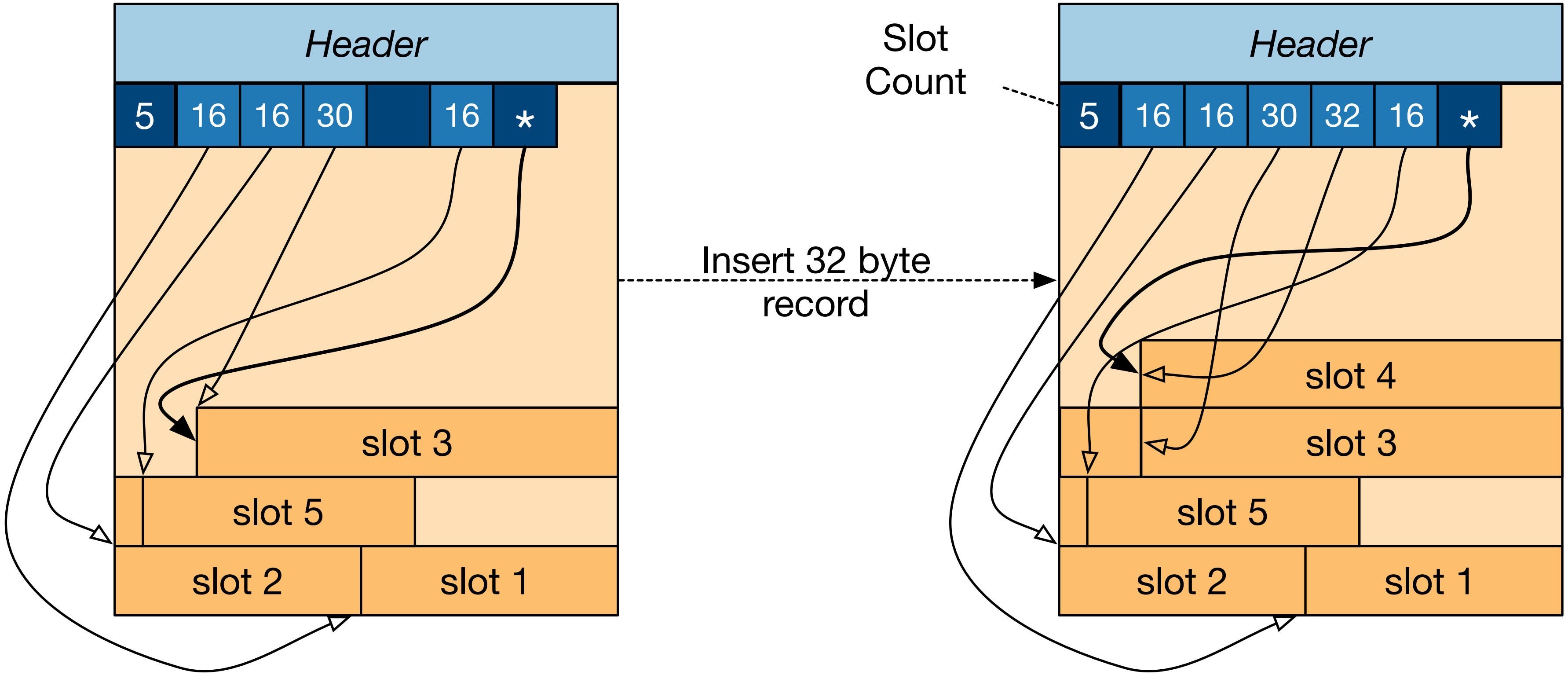
Record id = page id, slot location in directory



Set slot directory entry to null
Doesn't impact other records

Delete a record

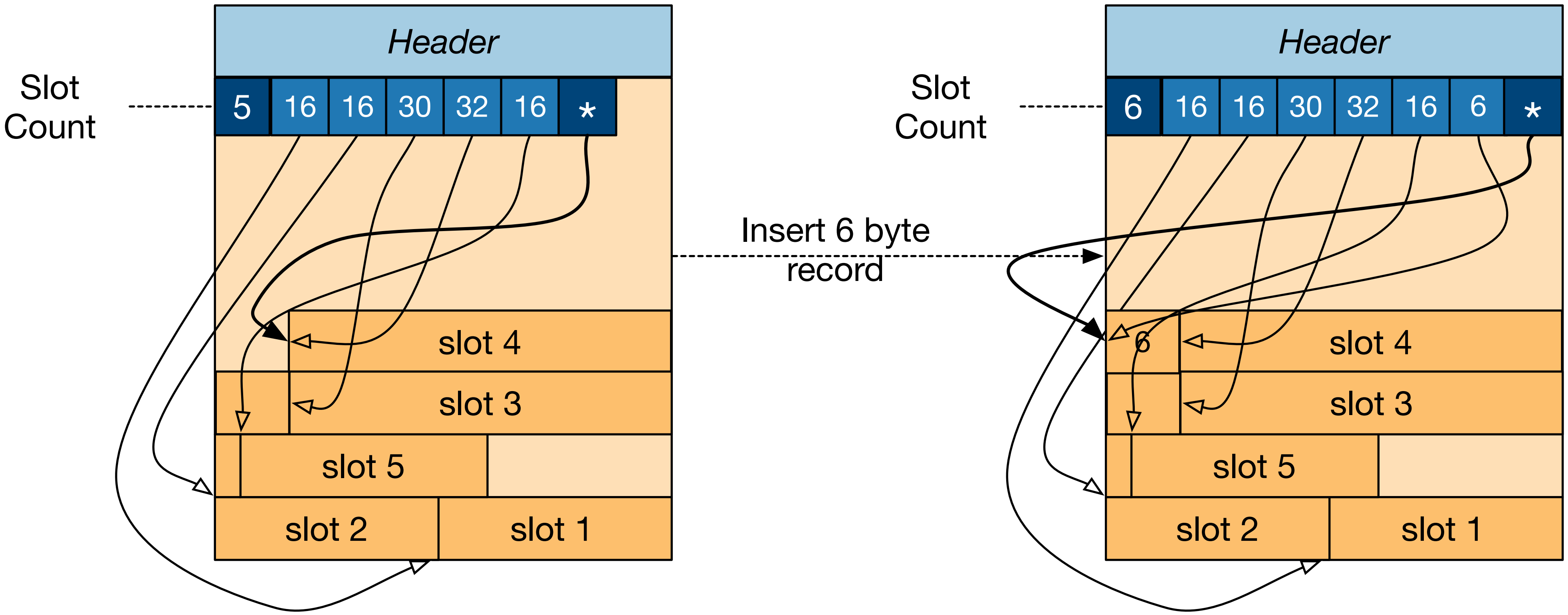
Insert a record



New record has rid
(page 5, slot 4)

Place record in free space
Reuse available free slots
Update the free space pointer

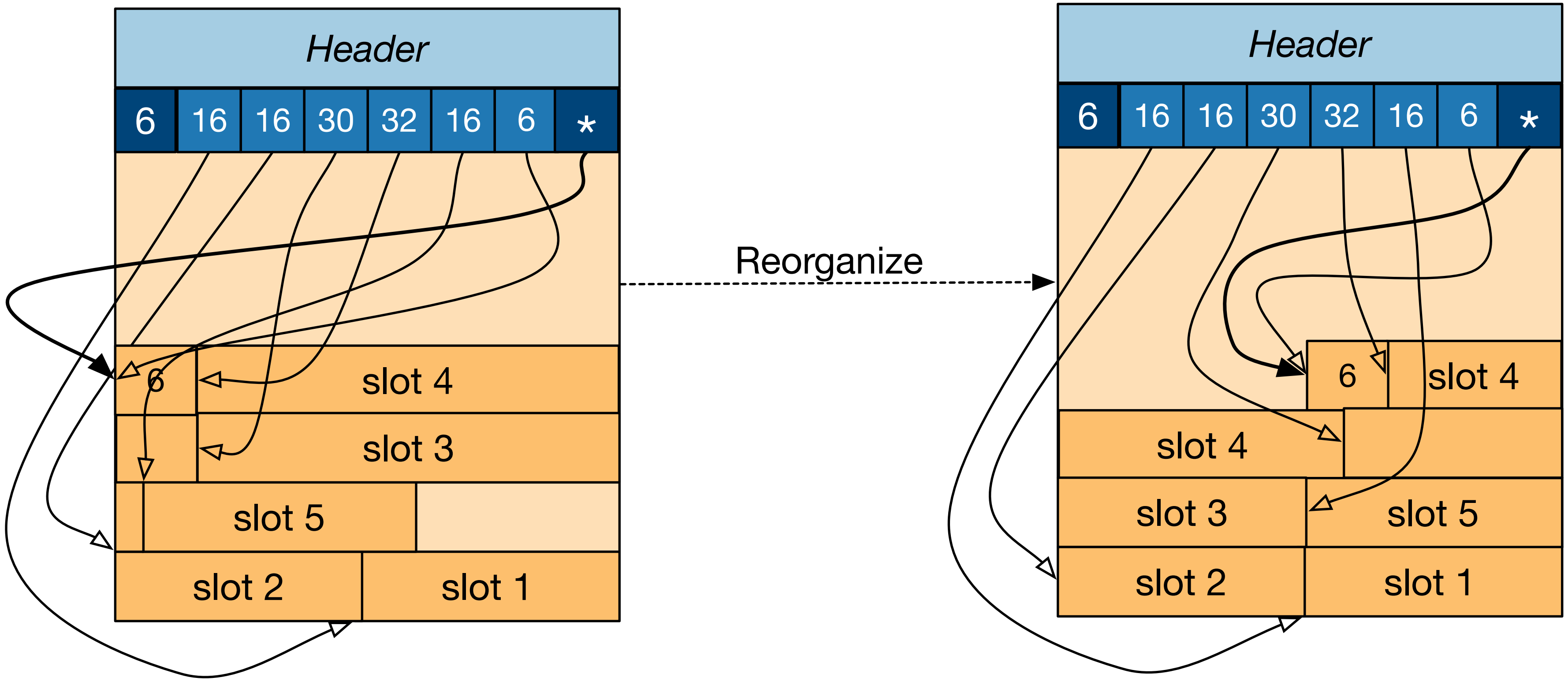
Insert a record – grow slots



New record has rid
(page 5, slot 6)

Place record in free space
Create new slot at the end & update slot count
Update the free space pointer

Reorganize page



Record location in
page changes but
not rids

Pays off to allow some degree of fragmentation

Compact records on the page but you don’t need to change a record’s slot #!

File Organizations

A Database File

A Database (Logical) File

A collection of pages, each containing a collection of records. A file can be a table or an index.

Could span multiple physical OS files or even devices.

Supports an API for

- Insert, delete or modify record
- Fetch by record id (page #, slot#)
- Scan all records (with filter condition)

A Heap File

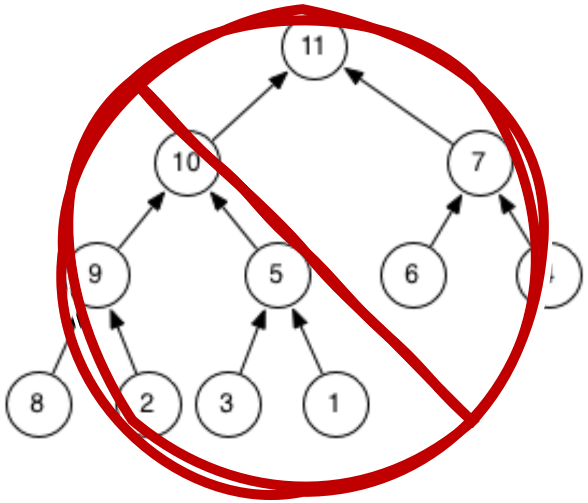
Collection of records in *no* specific order

Can grow or shrink by allocating or deallocating page.

Supports record level operations by keeping track of:

- the pages in a file
- the free space on pages
- the records on a page

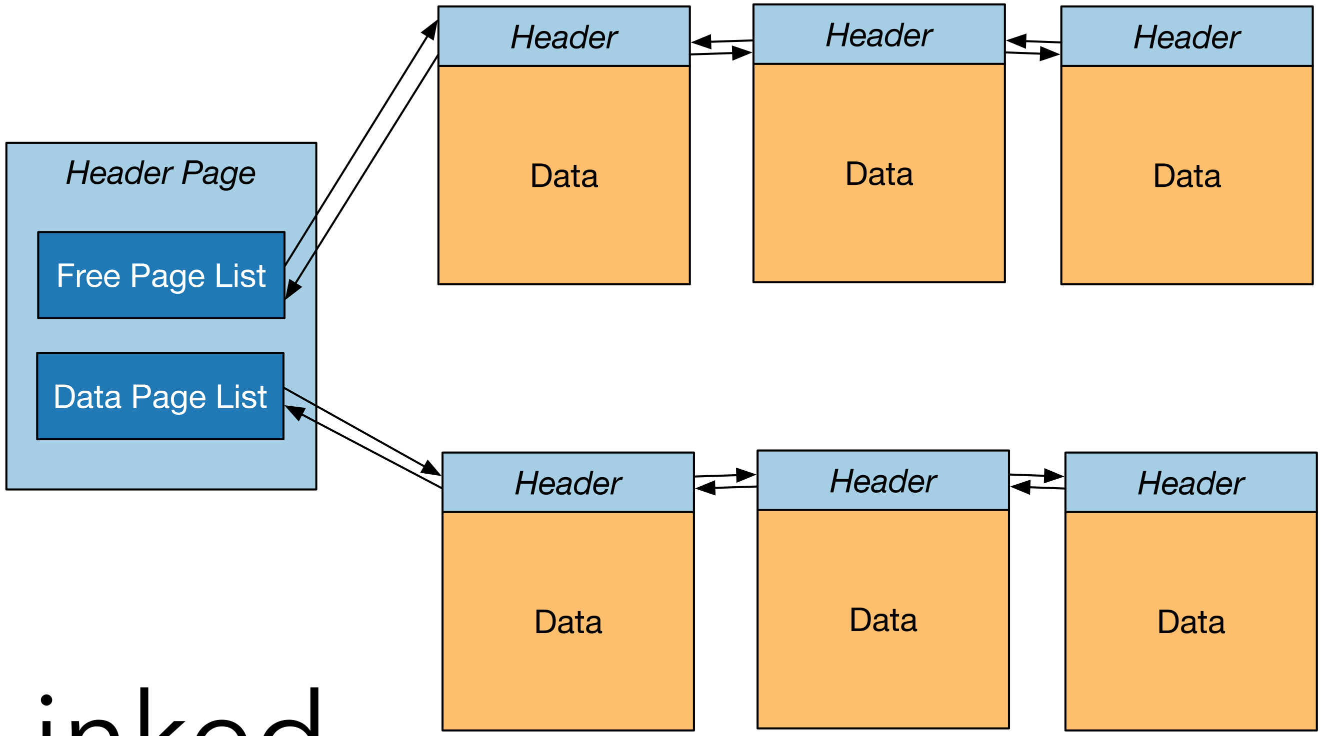
The ***catalog*** keeps track of a heap's header page ID and heap file name (i.e., the mapping from a table to the heap file)



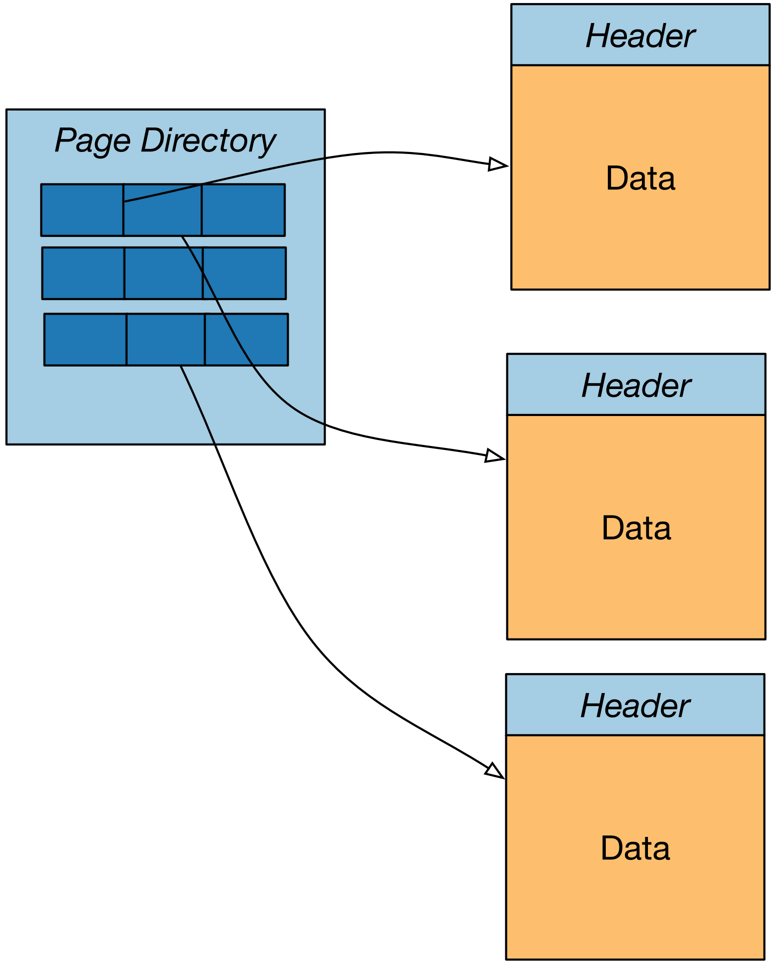
Option 1: Linked Listed Heap File

To find a specific page, you must sequentially scan the lists.

Each page keeps track of the number of free slots in itself.



How do I find page 5? How do I find enough space for a record of length *x* bytes?



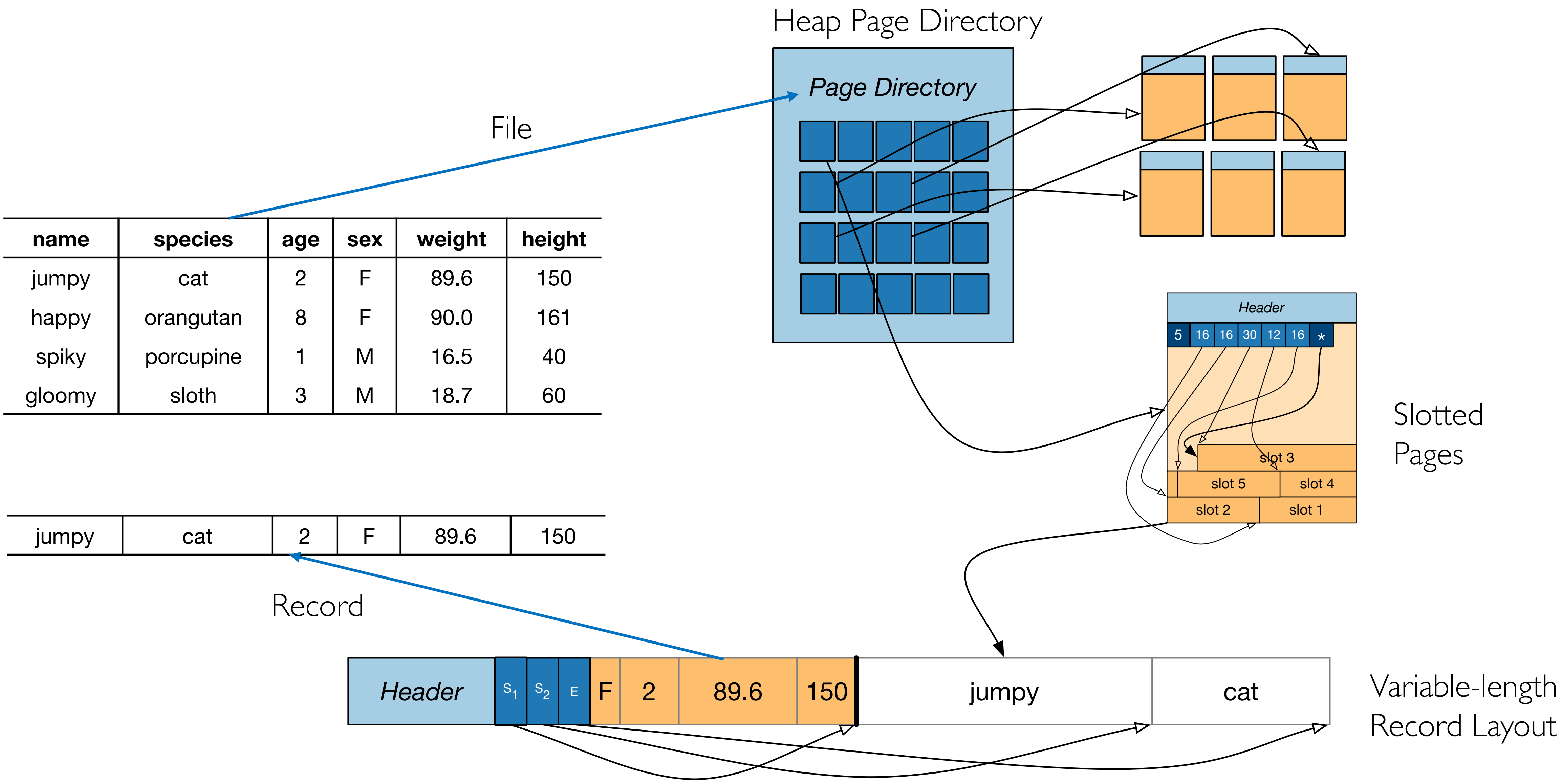
One more level of indirection: Use the page-directory to find the location of page p.

Each directory entry also includes additional information like # of free bytes/slots in a page.

Cache page directory!

Option 2: Heap File Page Directory

All together now



name	species	age	sex	weight	height
jumpy	cat	2	F	89.6	150
happy	orangutan	8	F	90.0	161
spiky	porcupine	1	M	16.5	40
gloomy	sloth	3	M	18.7	60

jumpy	cat	2	F	89.6	150

Unordered Heap Files

- Records placed arbitrarily across pages

Clustered Heap Files

- Records and pages are grouped

Sorted Files

- Pages and records are in sorted order

Index Files

- B+ Trees, Linear Hashing, …
- May contain records or *point to records in other files*

Other Organizations

How to choose the best file organization?

Identify your access patterns/workloads

Is it a read-mostly workload, or write-heavy? Are there many equality-searches? range-searches? full scans?

Create a model to quantify your trade-offs

- Estimate in a principled way the costs. *Crude & insightful* (not complex & perfect)
- Identify assumptions upfront

How to choose the best file organization?

Identify your access patterns/workloads

Create a model to quantify your trade-offs

	Heap File	**Sorted**
Scan		
Equality-search		
Range-search		
Single Record Insert		
Single Record Delete		

Heap File vs. Sorted

Identify your access patterns/workloads

Create a model to quantify your trade-offs

Heap File vs. Sorted

Estimate in a principled way the costs.

Crude & insightful (not complex & perfect)

- Time to read or write a block/page (T)
- Number of blocks/pages (B)
- Number of records per page (R)
- Conduct *average-case* analysis

Identify assumptions upfront

- Heap files append inserts
- Sorted files are packed (always compact after deletion)
- Sorted files are sorted by search key
- Ignore sequential vs. random IO, in-memory costs, …

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search		
Range-search		
Single Record Insert		
Single Record Delete		

Cost of a scan

Cost of an equality-search

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search	Is it $B \times T$ also?	*Does sorting help?*
Range-search		
Single Record Insert		
Single Record Delete		

What is the probability $P(i)$ that I find the record with the search key on page i?

Assume uniformly random keys

$$P(i) = \frac{1}{B}$$

How many pages will I visit on average to hit the search key?

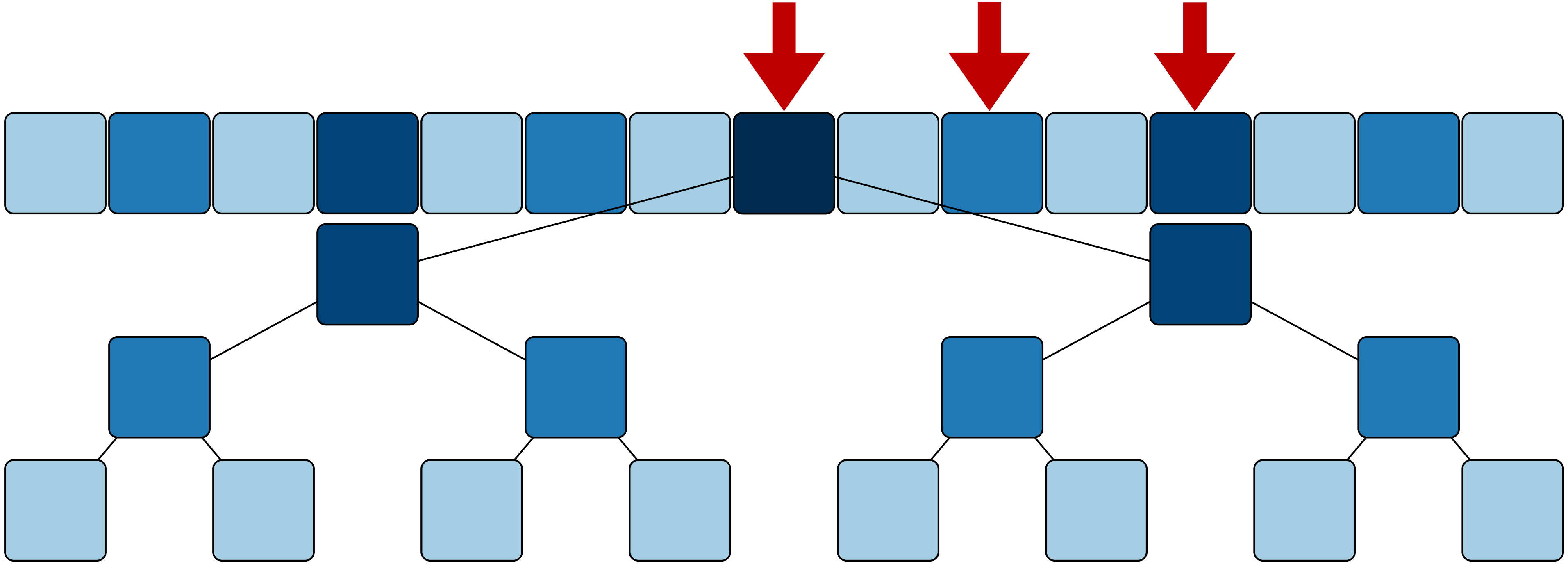
If you start from beginning to end scanning all pages, at each page there is 1/B chance that it has the search key, and you stop the scan.

$$\sum_{i=1}^{B} i \times P(i) = \sum_{i=1}^{B} i \frac{1}{B} \approx \frac{B}{2}$$

Cost of an equality-search – Heap File

Cost of an equality-search – Sorted File

Do a binary search on the sorted file.



$P(i)$	IOs so far
$\frac{1}{B}$	1
$\frac{1}{B/2} = \frac{2}{B}$	2
$\frac{1}{B/4} = \frac{4}{B}$	3
$\frac{1}{B/8} = \frac{8}{B}$	4
$\frac{2^{i-1}}{B}$	i
$\frac{2^{\log_2 B-1}}{B}$	$\log_2 B$

$$\sum_{i=1}^{\log_2 B} i\,\frac{2^{i-1}}{B} = \log_2 B - \frac{(B-1)}{B} \approx \log_2 B$$

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search	${}^{B}/_{2} \times T$	$\log_2 B \times T$
Range-search		
Single Record Insert		
Single Record Delete		

Cost of an equality-search

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search	${}^{B}/_{2} \times T$	$\log_2 B \times T$
Range-search	Do we find every value in the range?	Find smallest value & scan right
Single Record Insert		
Single Record Delete		

Cost of a range-search - key in [x, y]

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search	$B/_{2} \times T$	$\log_2 B \times T$
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$
Single Record Insert		
Single Record Delete		

Cost of a range-search

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search	${}^{B}/_{2} \times T$	$\log_2 B \times T$
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$
Single Record Insert	Append at the end	Find, insert, shift
Single Record Delete		

Cost of an insert

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search	${}^{B}/_{2} \times T$	$\log_2 B \times T$
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$
Single Record Insert	$2T$	$\left(\log_2 B + 2\,{}^{B}/_{2}\right) \times T$
Single Record Delete		

Cost of an insert

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search	$B/_2 \times T$	$\log_2 B \times T$
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$
Single Record Insert	$2T$	$(\log_2 B + B) \times T$
Single Record Delete	Find, delete, write	Find, delete, shift

Cost of a delete

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search	${}^{B}/_{2} \times T$	$\log_2 B \times T$
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$
Single Record Insert	$2T$	$(\log_2 B + B) \times T$
Single Record Delete	$({}^{B}/_{2} + 1) \times T$	$(\log_2 B + B) \times T$

Cost of a delete

	Heap File	**Sorted**
Scan	$B \times T$	$B \times T$
Equality-search	${}^{B}/_{2} \times T$	$\log_2 B \times T$
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$
Single Record Insert	$2T$	$(\log_2 B + B) \times T$
Single Record Delete	$({}^{B}/_{2} + 1) \times T$	$(\log_2 B + B) \times T$

A workload may have a different distribution of each of these tasks!

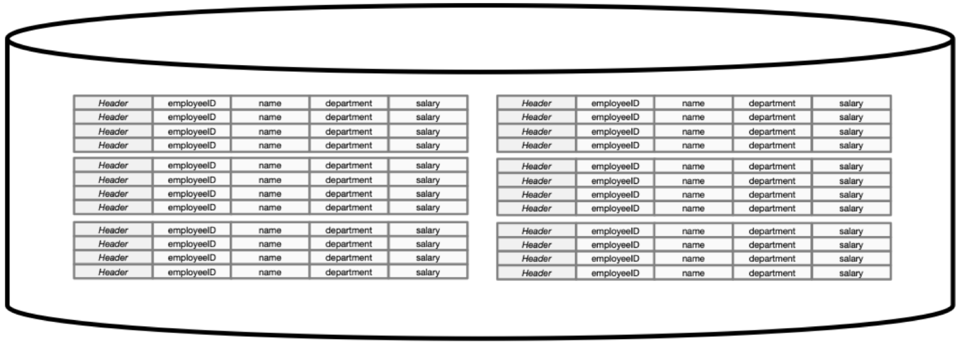
What happens when we change some of our assumptions?

We can do better with indexes

Heap File vs. Sorted

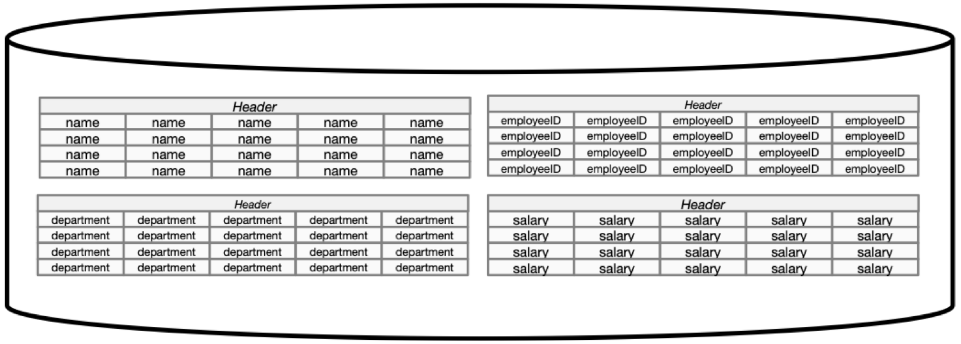
Row Stores vs. Column Stores

Row Stores



Header	employeeID	name	department	salary
Header	employeeID	name	department	salary
Header	employeeID	name	department	salary
Header	employeeID	name	department	salary

Column Stores



Header				
salary	salary	salary	salary	salary
salary	salary	salary	salary	salary
salary	salary	salary	salary	salary
salary	salary	salary	salary	salary

When to use Row-Stores?

THE GOOD

- Good performance for writes (inserts, updates, deletes) – transactional workloads
- Good for queries that need the entire tuple.

THE BAD

- Not ideal for scanning large portions of only subsets of a table's attributes.

When to use Column-Stores?

THE GOOD

- Ideal for analytical workloads where read-only queries perform large scans over a subset of the table's attributes.
- Better data compression
- Easy schema expansion: add a new column doesn't entail rewrite
- Same column can be replicated with different orders

THE BAD

- Slow for inserts, updates, deletes --- tuple stitching or joining overhead.