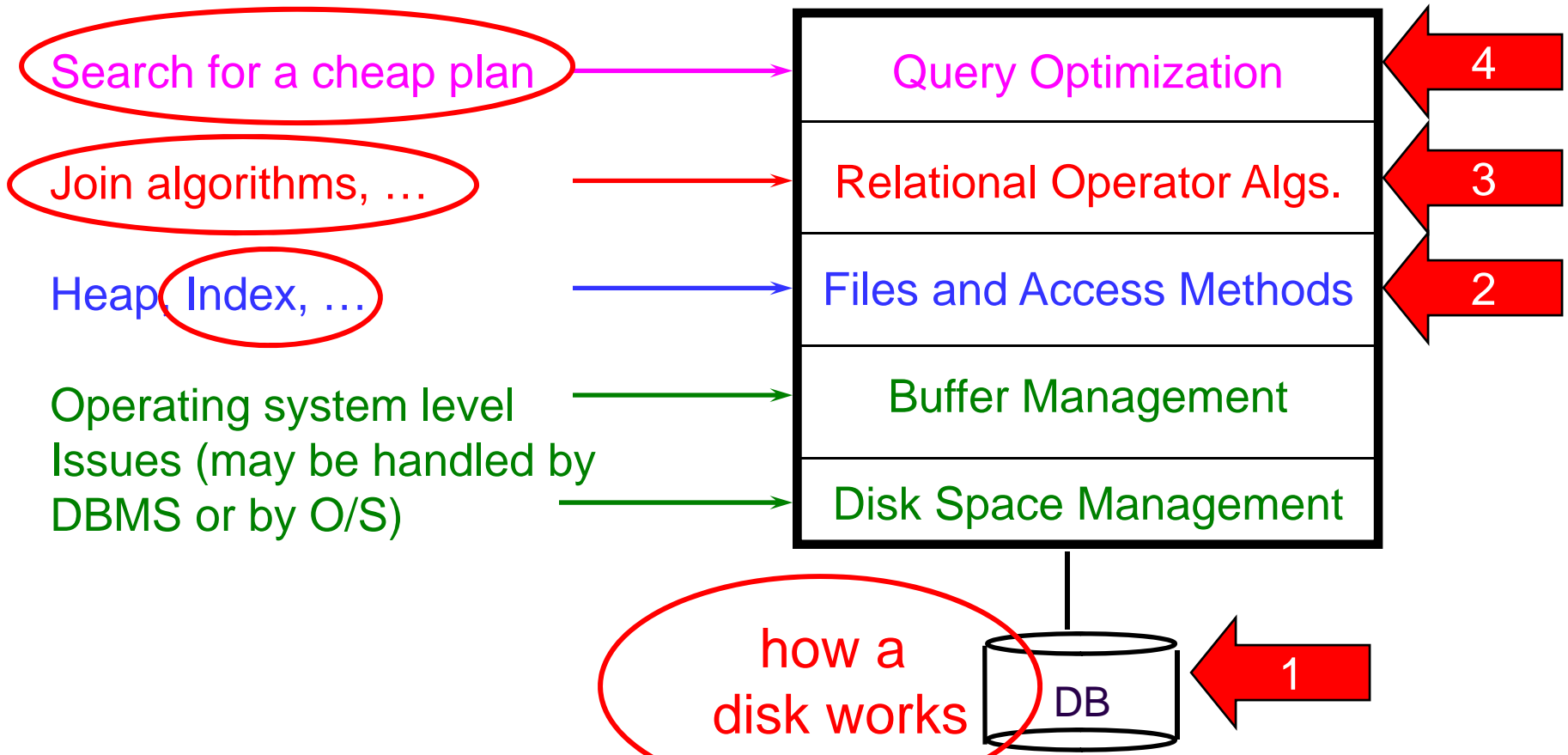


Query Optimization

- Which query plan is the fastest?
- How many query plans are there?
- How can we estimate the cost of a plan?
- But wait, how are queries (query operators) implemented?
- But wait, how are the files stored?

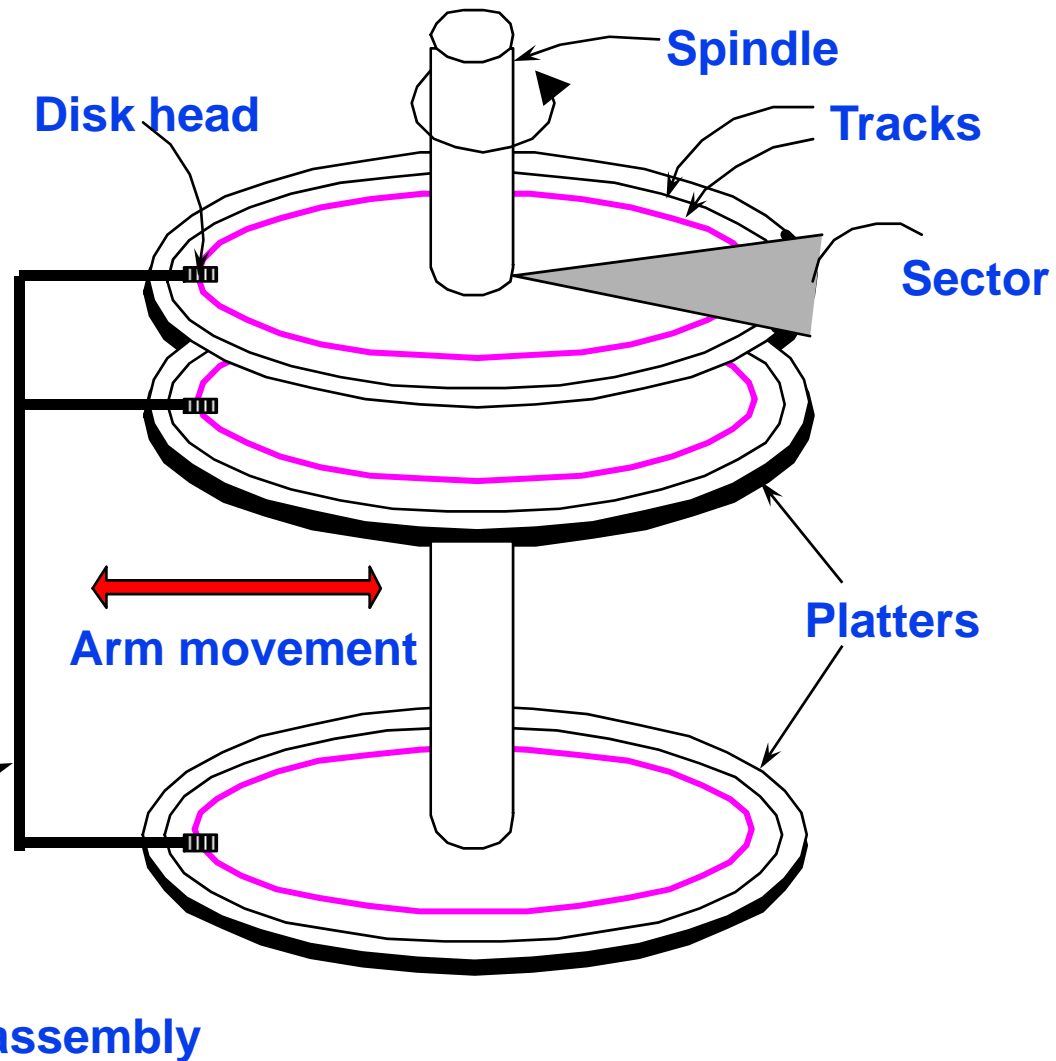
We'll just introduce these ideas and we'll start from bottom

Relational Algebra Query Tree



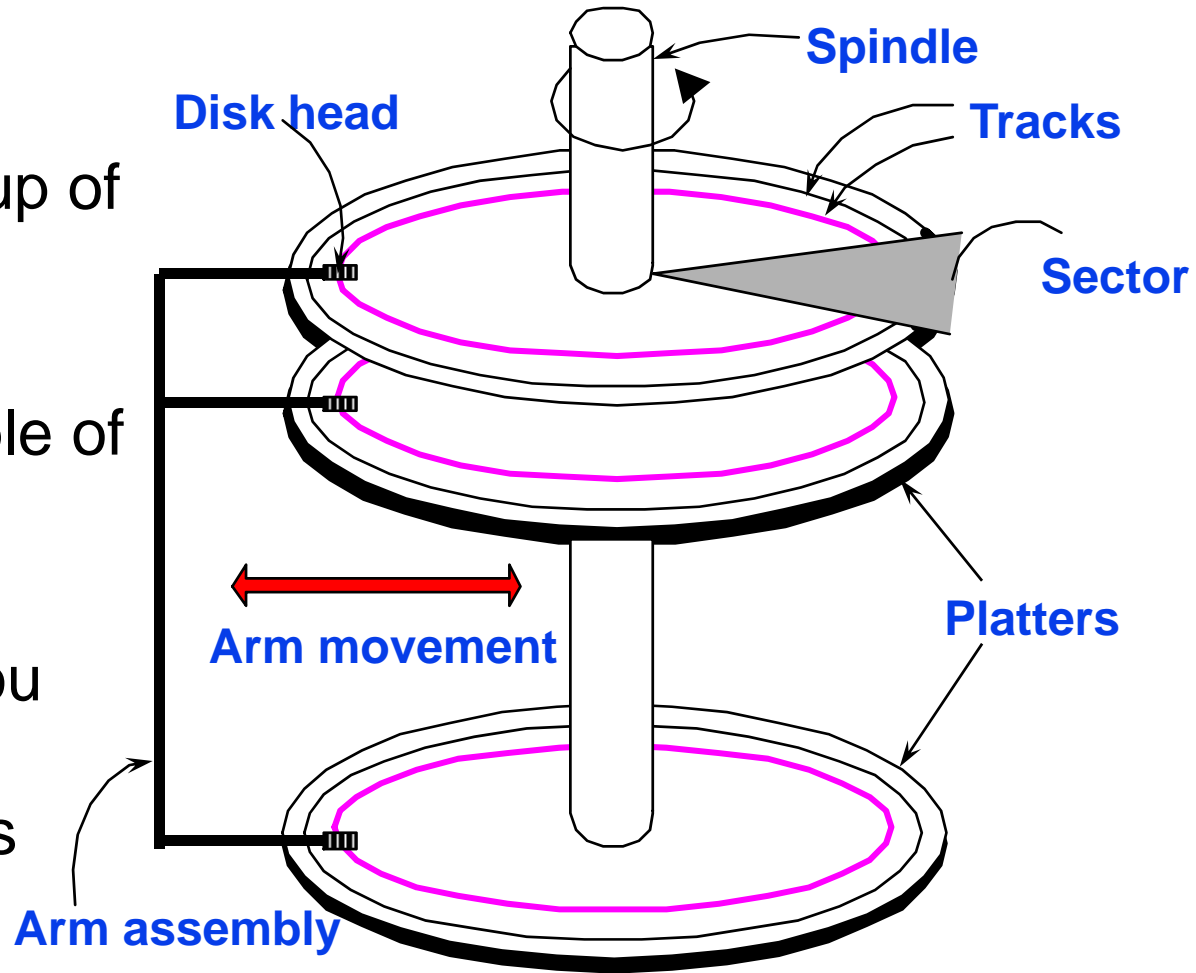
Components of a Disk

- platters are always spinning (say, 120rps).
- one head reads/writes at any one time.
- to read a record:
 - **position arm (seek)**
 - engage head
 - **wait for data to spin by**
 - **read (transfer data)**



More terminology

- ❖ Each **track** is made up of fixed size **sectors**.
- ❖ **Page size** is a multiple of **sector size**.
- ❖ All the tracks that you can reach from one position of the arm is called a **cylinder** (imaginary!).



Cost of Accessing Data on Disk

- Time to access (read/write) a disk block:
 - *seek time* (moving arms to position disk head on track)
 - *rotational delay* (waiting for block to rotate under head)
 - *transfer time* (actually moving data to/from disk surface)
- Key to lower I/O cost: **reduce seek/rotation delays!**
(you have to wait for the transfer time, no matter what)
- **Query cost is often measured in the number of page I/Os – often simplified to assume each page I/O costs the same**

Disk: 10^5 to 10^6 times slower than memory (we'll use in this 10^6 example)

- Disk access time (all three costs together) is about 2 to 7 milliseconds
- Memory access time: 50 to 70 nanoseconds
- 7 milliseconds vs. 70 nanoseconds
therefore disk access is 100,000 times slower than memory access
- **Contrast 1 second** (pick up a piece of paper) **vs. 100,000 seconds** (drive to SF and back – about 28 hours)

Block (page) size vs. record size

- **Page** –smallest unit of transfer supported by OS
- **Block** – Multiple of page, smallest unit of transfer supported by an application or a disk volume.
- Block and page are often used interchangeably.
- “typical” record size ... maybe a few hundred up to few thousand bytes
- “typical” page size 4K, 8K
- When would we choose block size to be **larger**?
- When would we choose block size to be **smaller**?

Real-life Indexes

- What will the user enter to search for records?
 - Librarian using a Library Catalog
 - Clerk in a video store
 - Receptionist in a medical office
- The data value that the user enters (to look something up) is called a **search key**

Database Indexes

- Given Emp(ID, name, age, address)
- What are the possible search keys?

Note:

- You can build an index on any subset of the fields of a table.
- You can build more than one index for the same table.
- “*Search key*” is *not* the same as a *key* for the table. Values of a “search key” need not be unique.

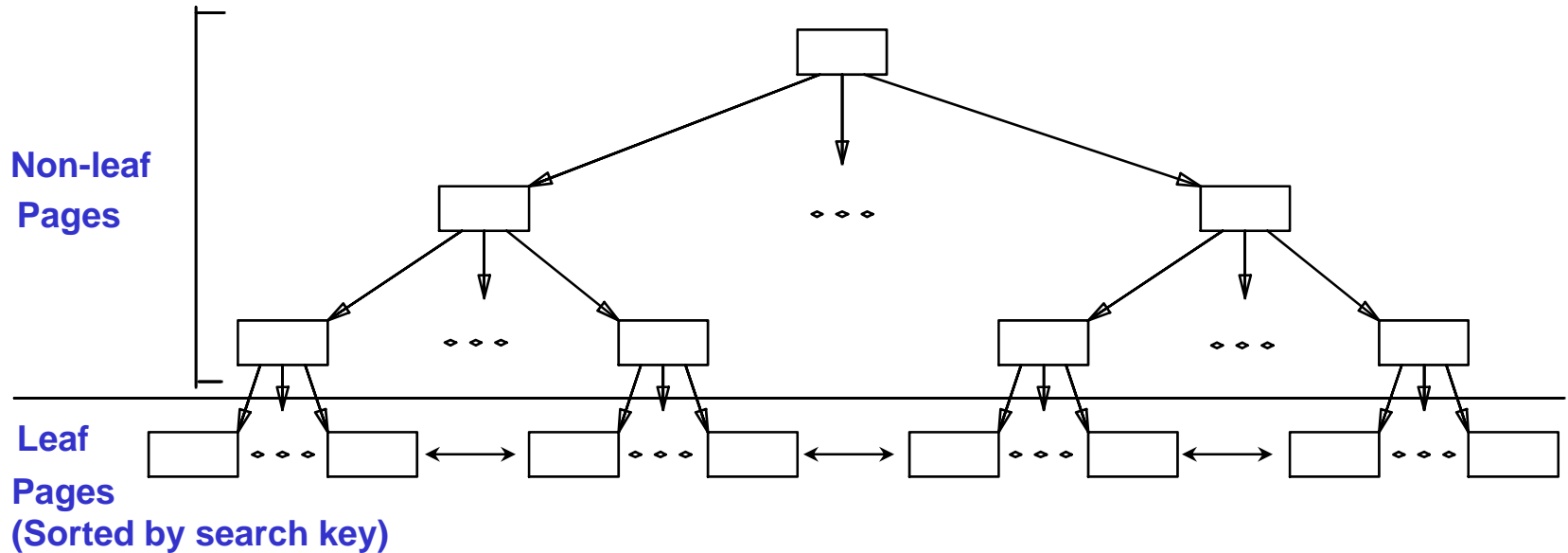
Index for a File

- An *Index* is a data structure that speeds up selections on the *search key field(s)*
- An index transforms a search key k into a *data entry k^** .
- Given k^* , you can get to the record(s) with the search key k in one I/O.

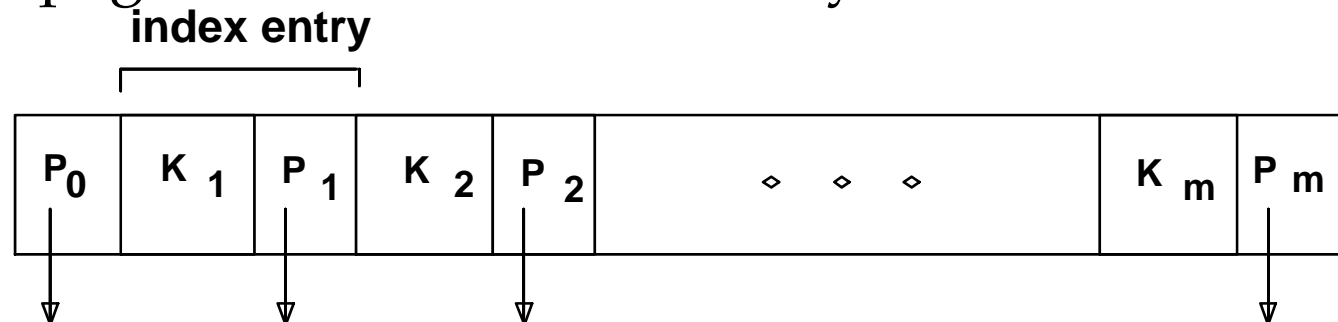
Most Indexes are Tree Structured

- Tree-structured indexes support *range searches* and *equality searches*.
 - *ISAM*: static structure (old technology)...index is built just once, when the file is loaded. Uses overflow areas, so the tree can become very unbalanced.
 - *B+ tree*: dynamic – index is adjusted as records are inserted and deleted in the file. Index remains balanced.

B+ Tree Indexes



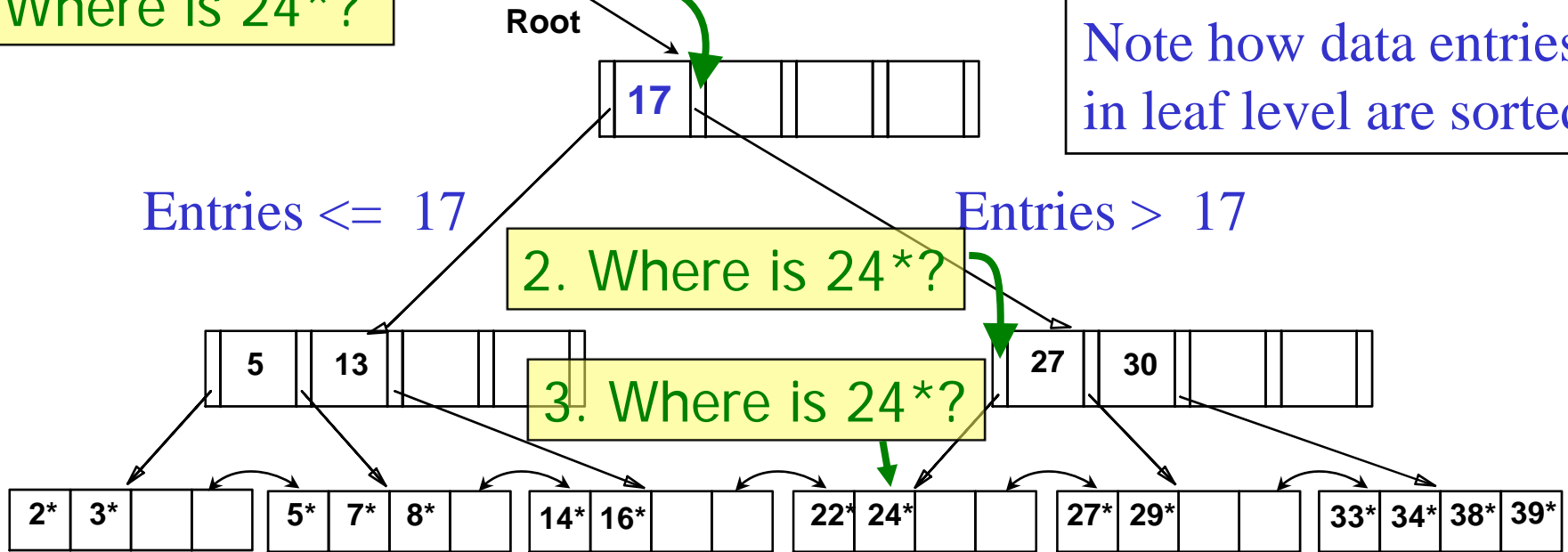
- ❖ Leaf pages contain *data entries*, and are chained (prev & next)
- ❖ Non-leaf pages have *index entries*; only used to direct searches:



1. Start at top:
Where is 24*?

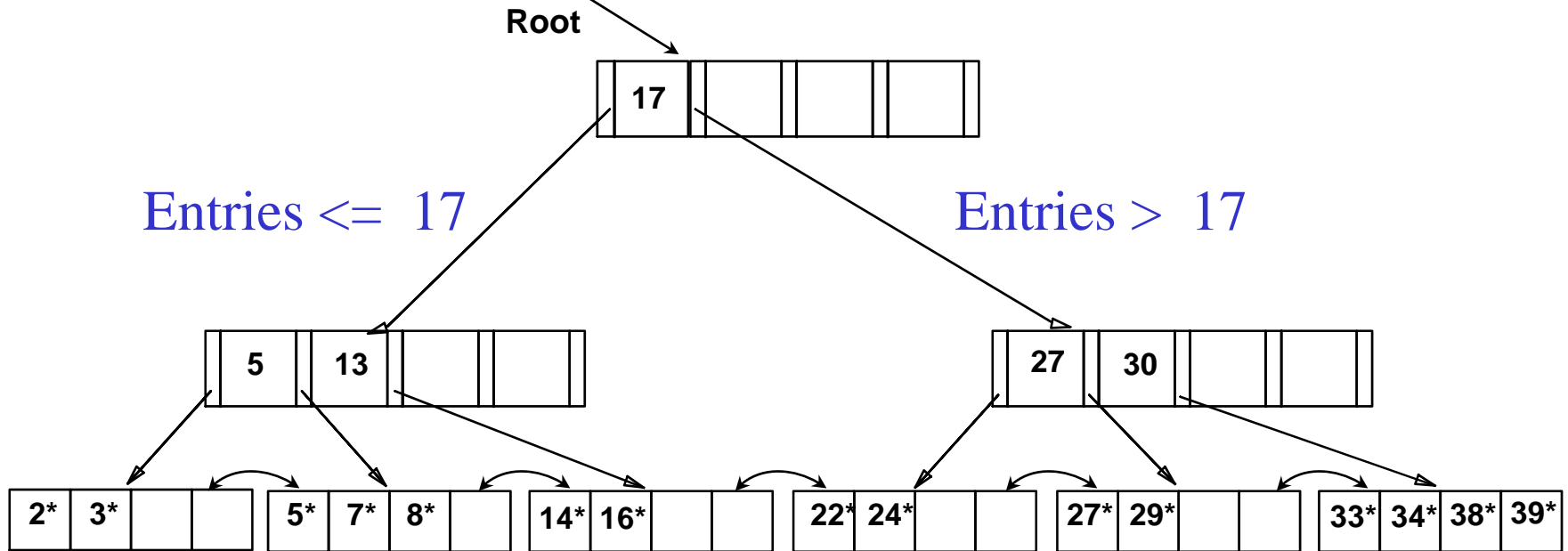
Search a B+ Tree

Note how data entries in leaf level are sorted



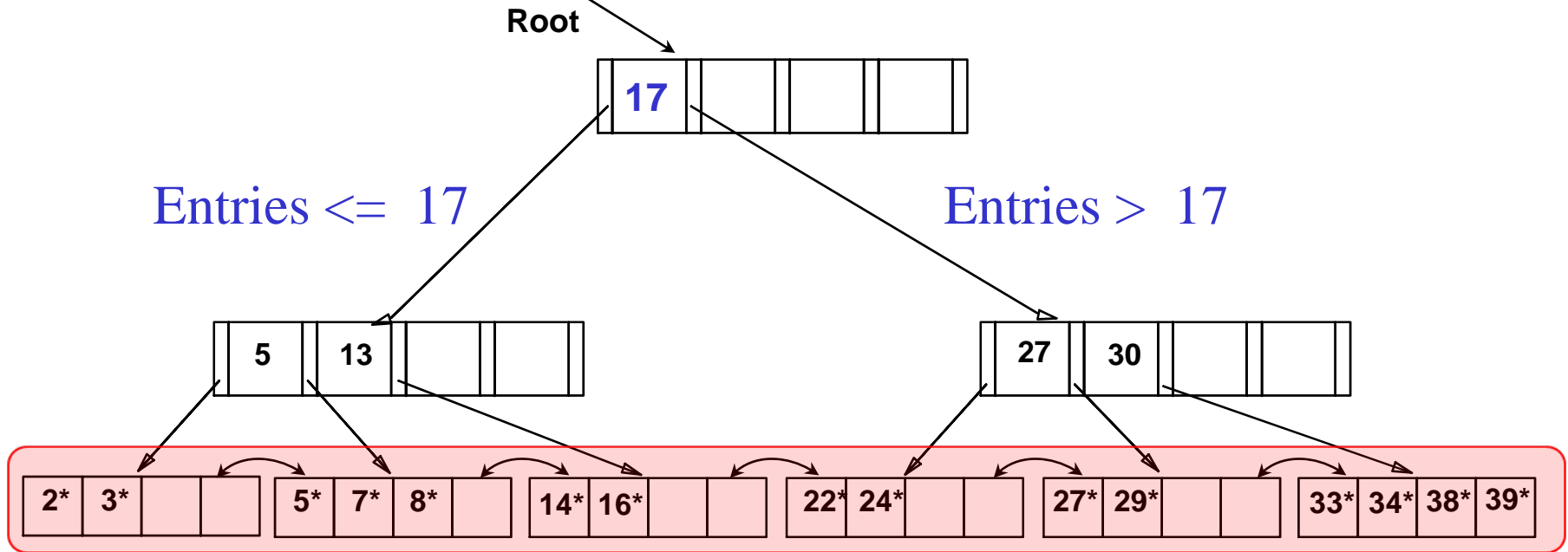
- Find 24* (shown above). How many pages (I/Os)?
- Find 28* How many pages (I/Os)?
- Find 29* How many pages (I/Os)?
- Find all data entries $> 15^*$ and $< 30^*$
How many pages (I/Os)?

Updates in a B+ Tree



- Insert/modify/delete: Find data entry in leaf, change it.
- When inserting, if your page is full, then SPLIT your page; then add one entry to your parent.
 - And change sometimes bubbles up the tree (if parent is full)
 - This keeps the tree *balanced*: each data retrieval takes the same number of I/Os
- If you combine pages on delete, each page is $>$ half full.

Example B+ Tree



- Data entry pages are chained together – doubly-linked list
- Two styles for the leaf level of an index:
 - The leaf-level contains the data record directly.
In this case, the data records are sorted, physically.
 - The leaf-level contains data entries (keyval, ptr to actual record)
- The books call these “Alternative 1 and Alternative 2”

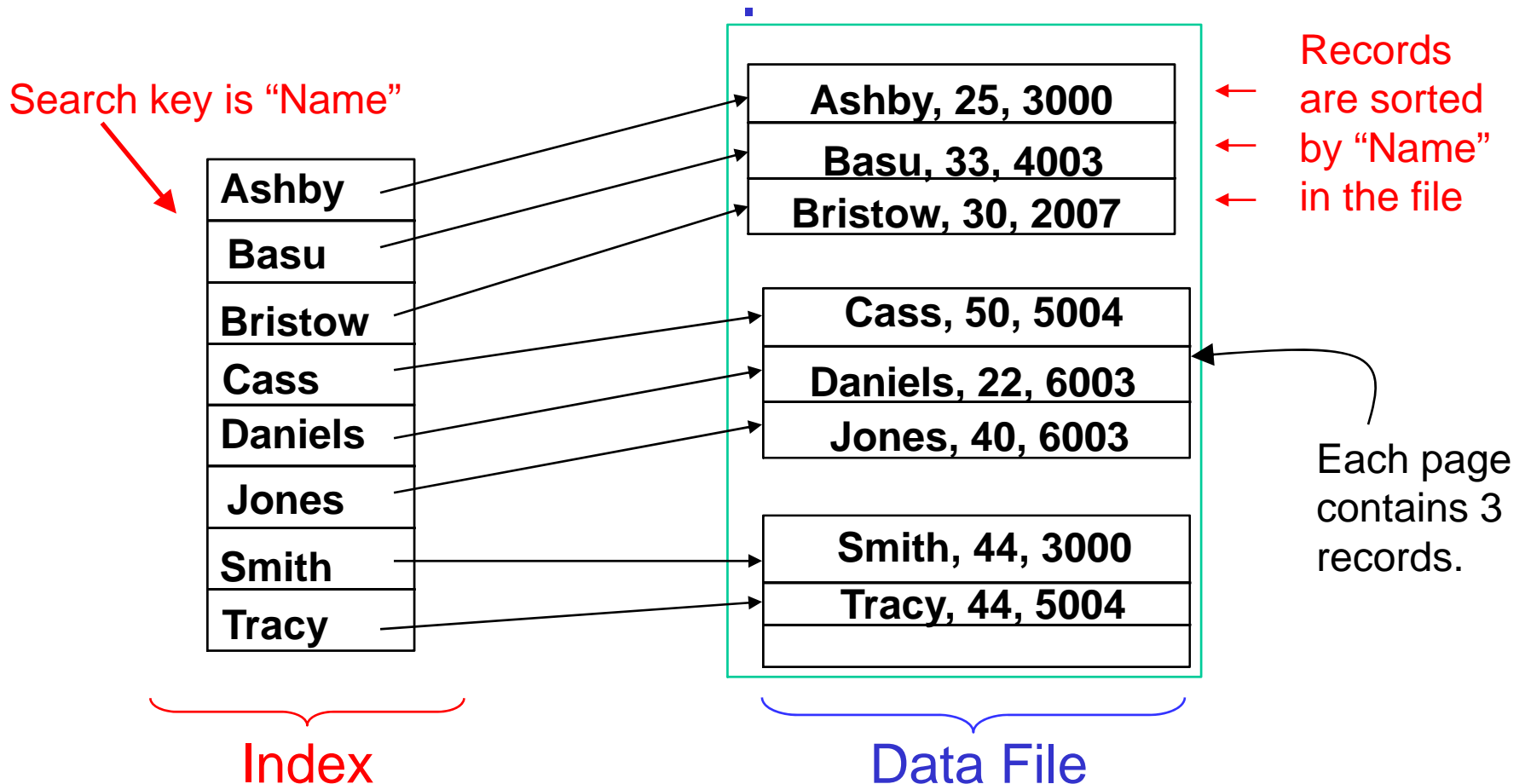
Costs of an Index

- If you define an index in your database, you will incur three **costs**
 - Space to store the index
 - Updates to the search key will be slower
 - The optimizer will take longer because it has more choices
- There is one **advantage** to having an index
 - Some queries run faster (better be sure about this)

Dense: One index entry for each data record

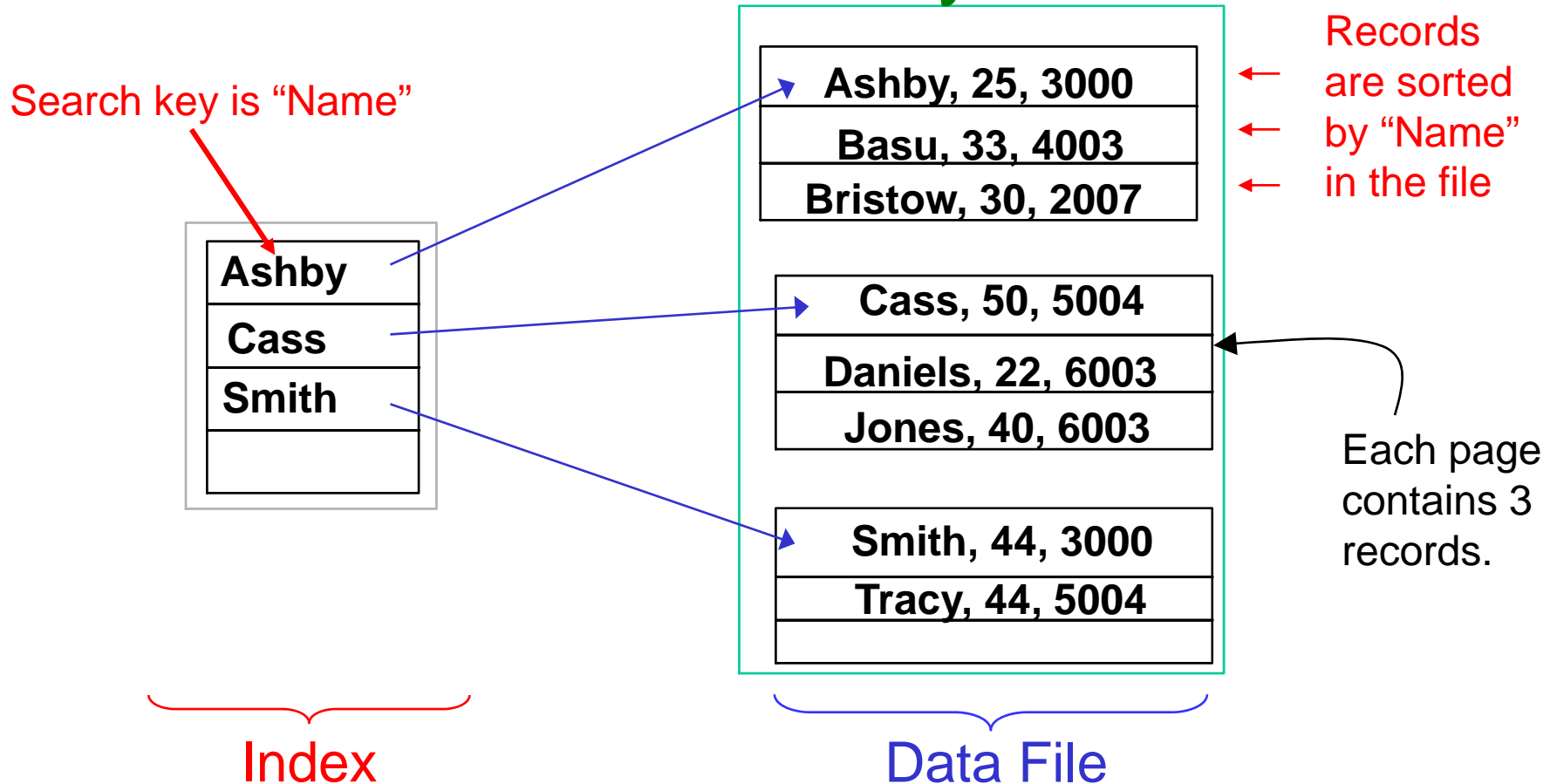
Clustered Index:

Records are sorted based on search key.



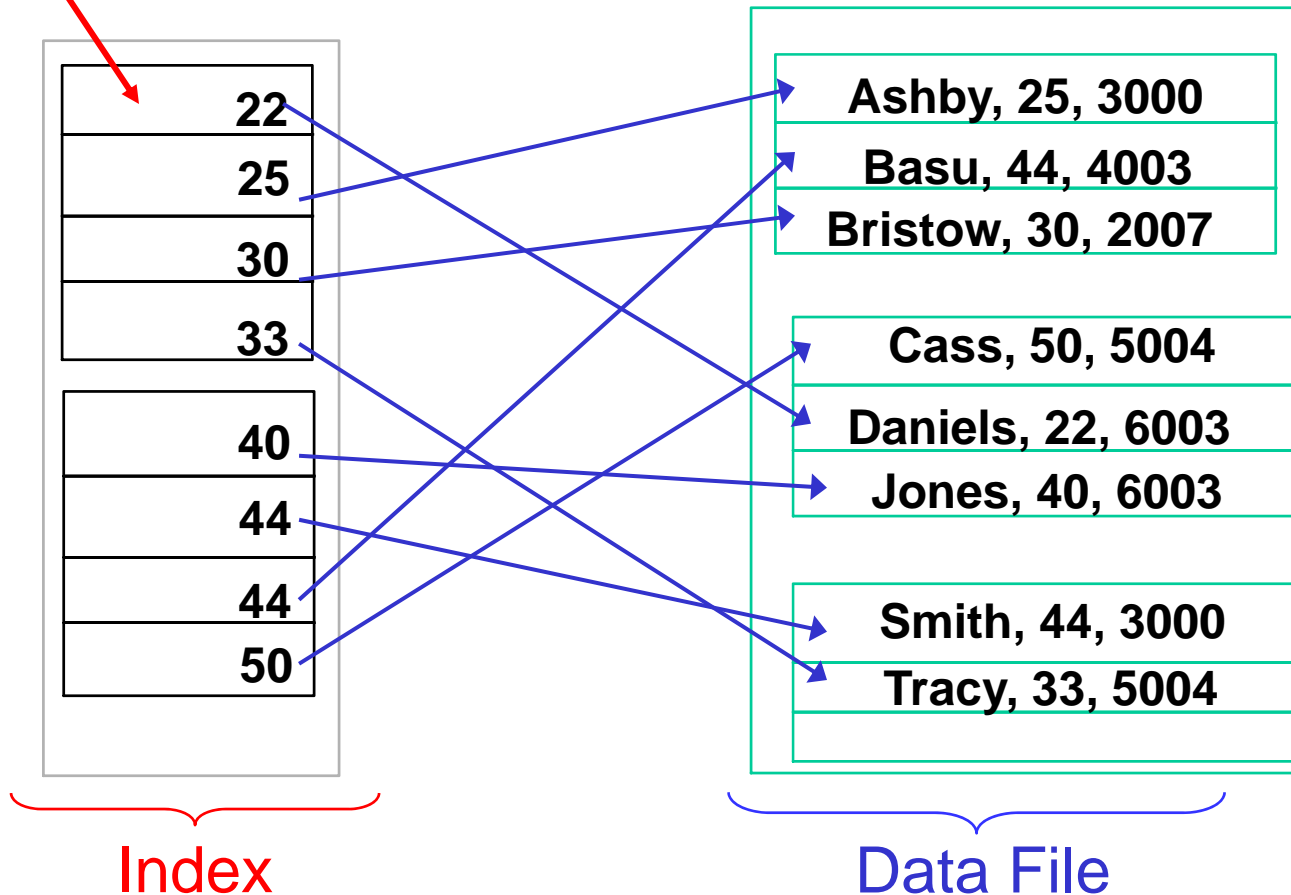
Sparse: 1 index entry per PAGE of data

Clustered Index: data is sorted on search key

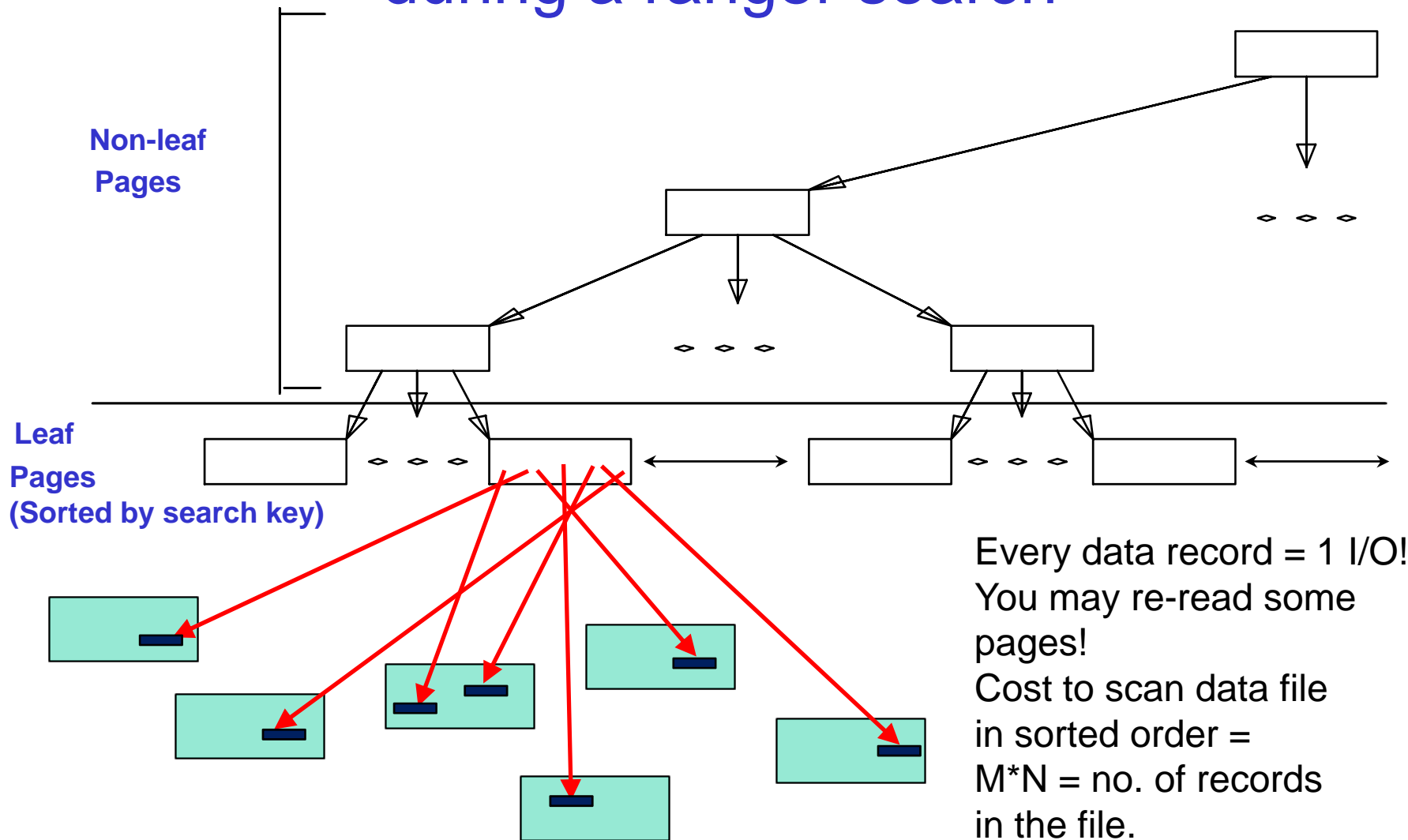


Unclustered Index: Records NOT Sorted on Search Key. It must be dense!

Search key is "Age"



I/Os using a dense, unclustered index during a ranger search



Clustered vs. Non-clustered Index

- Consider a telephone book as an index to telephone numbers
 - What is the primary search key?
 - Is it a clustered or unclustered index?
 - Is it a dense or sparse index?
 - Can you find a range of entries using this index?
- Imagine we have an unclustered index for a phone book based on street address
 - Can you efficiently find a range of entries – a range of addresses?

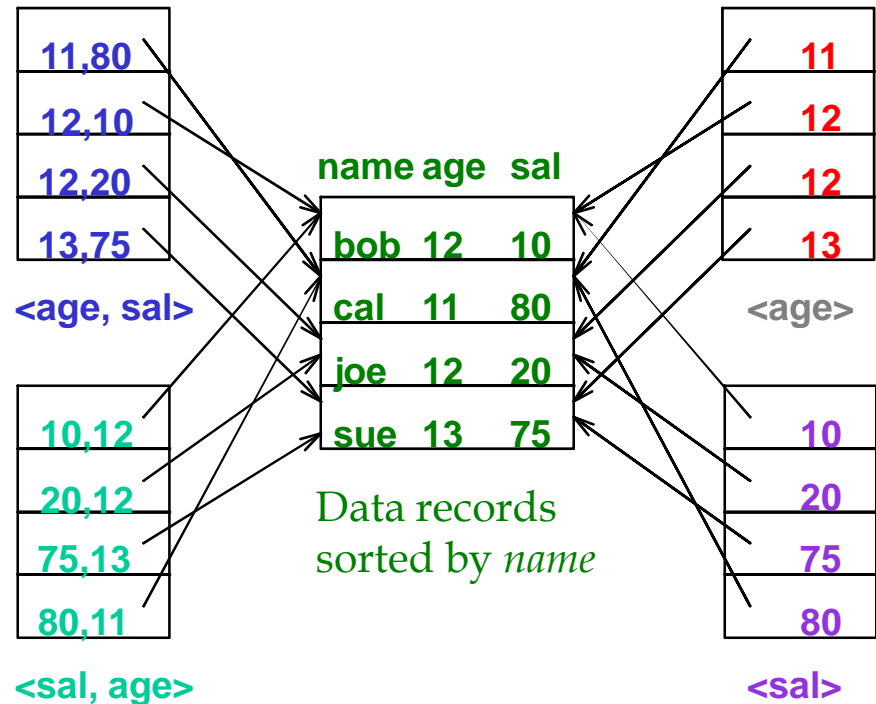
Using Composite Search Keys

Which indexes can you use for each of these queries?

You can use index when the answer is one contiguous block of data in the data entry level

- age = 12
- age = 12 and sal = 20
- age=12 and sal > 10
- age > 12 and sal > 30

Which of these indexed are clustered?



Indexes

- DBMSs often create a clustered index on all primary keys.
- Note: primary keys are the values that must be used in foreign keys.
- Only **one clustered index** per table! Why?
- You need to decide whether you want additional (unclustered/secondary) indexes.
- You need to decide if you want composite indexes.

Join Algorithms – an Introduction

```
SELECT *  
FROM Reserves R1, Sailors S1  
WHERE R1.sid=S1.sid
```

$R \bowtie S$ is very common! And $R \times S$ followed by a selection is inefficient. So we process joins (rather than cross product) whenever possible. Lots of effort invested in join algorithms.

Assume: M pages in R , p_R tuples per page, N pages in S , p_S tuples per page.

In our examples, R is Reserves and S is Sailors.

Simple Nested Loops Join

```
Join on  $i^{\text{th}}$  column of R and  $j^{\text{th}}$  column of S
foreach tuple r in R do
    foreach tuple s in S do
        if  $r_i == s_j$  then add  $\langle r, s \rangle$  to result
```

- For each **tuple** in the *outer* relation R, we scan the entire *inner* relation S, tuple by tuple.
 - Cost: $M + (p_R * M) * N = 1000 + 100 * 1000 * 500$ I/Os
 - **50,001,000 I/Os** $\approx 500,010$ seconds ≈ 6 days

We assume approximately 100 I/Os per second

$M = 1000$ pages in R, $p_R = 100$ tuples per page,

$N = 500$ pages in S, $p_S = 80$ tuples per page.

Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...

Memory Buffers:

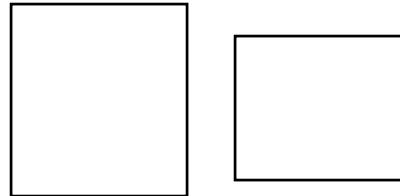


Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5

Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...



Memory Buffers:

2	...
12	...
6	...

...	2
...	13



Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5



Query Answer

2 2

Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...

Memory Buffers:

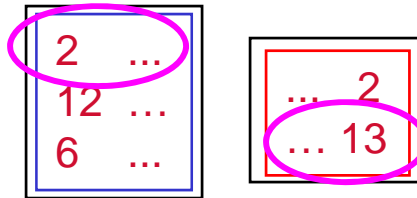


Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5

Query Answer

2 2

Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...

Memory Buffers:

2	...
12	...
6	...

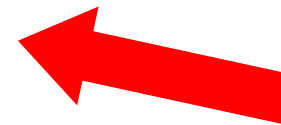
...	12
...	27

Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5



No match:
Discard!

Query Answer

2 2

Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...

Memory Buffers:

2	...
12	...
6	...

...	12
...	27

No match:
Discard!

Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5

Query Answer

2 2

Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...

Memory Buffers:

2	...
12	...
6	...

...	1
...	5

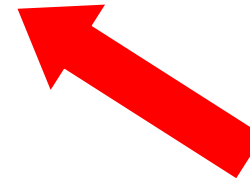
Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5

No match:
Discard!



Query Answer

2 2

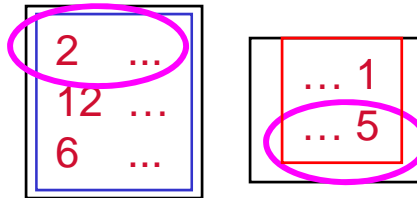
Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...

Memory Buffers:



No match:
Discard!

Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5

Query Answer

2 2

Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...

Memory Buffers:

2	...
12	...
6	...

...	2
...	13



No match:
Discard!

Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5

Query Answer

2 2

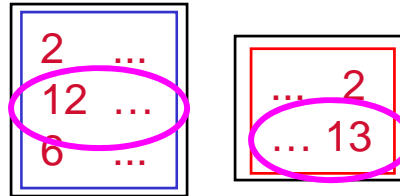
Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...

Memory Buffers:



No match:
Discard!

Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5

Query Answer

2 2

Simple Nested Loops Join

Table 1
on disk

2	...
12	...
6	...

1	...
5	...
27	...

Memory Buffers:

2	...
12	...
6	...

...	12
...	27

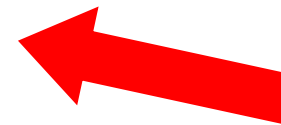
Match!

Table 2
on disk

...	2
...	13

...	12
...	27

...	1
...	5



Does this algorithm
work for

R1.sid < S1.sid?

Does this algorithm
work for cross
product?

Query Answer

2 2
12 12

And so forth ...