# CS150: Database & Datamining

## Lecture 10: File Organization & Index

ShanghaiTech-SIST

Spring 2019

*Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis.*

# Today's Lecture

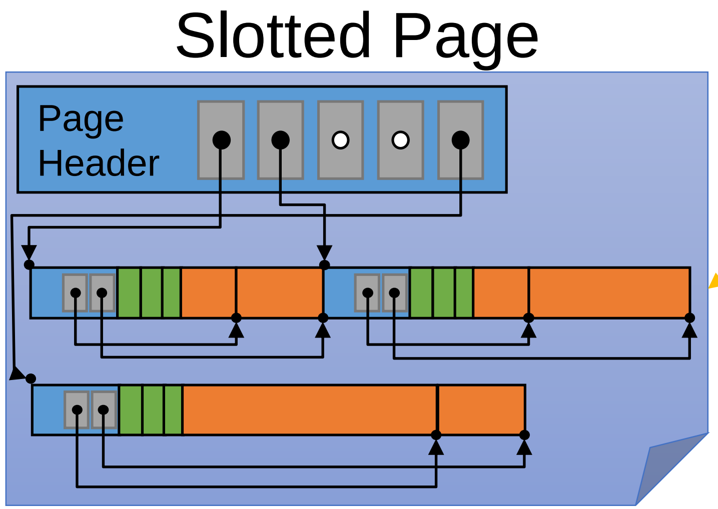
1. File Organizations & Cost Models

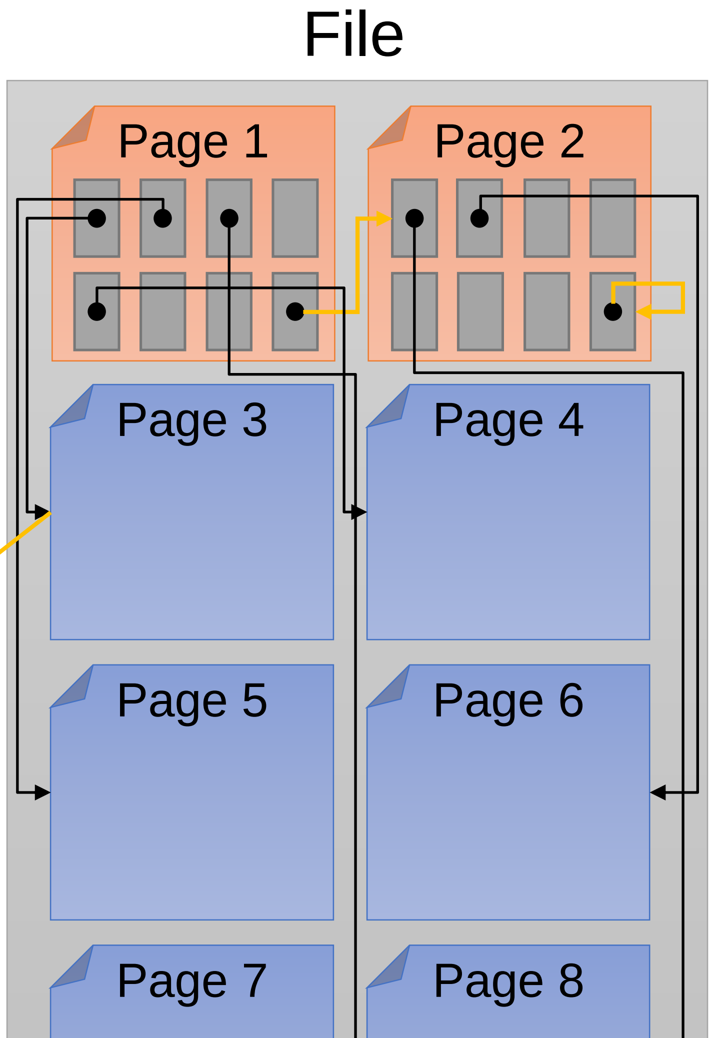
2. Indexes: Motivations & Basics

# 1. File Organizations

# How do we organize records and pages for certain operations?

- Ordering and grouping records



# Recall Heap Files

Heap Files

- Unordered collection of records
- Add/Remove records: Easy (Cost?)
- Scan: Easy (Cost?)
- Find a record?
  - Given a record_id: (File, Page, Slot)?
  - Matching username = ”joeyg”?

# Multiple File Organizations

Many alternatives exist, *each good in some situations and not so good in others*:

- Heap files: Suitable when typical access is a full scan of all records
- Sorted Files: Best for retrieval in *search key* order, or a **range** of records is needed
- Clustered Files & Indexes: Group data into blocks to enable fast lookup and efficient modification. (More on this soon …)

# Bigger Questions

- What is the "best" file organization?
  - Depends on access patterns…
  - how? what are they?
- Can we be quantitative about tradeoffs?
  - Better → How Much?

# Goals for Cost Analysis

- Big picture overheads for data access
  - We’ll (overly) simplify things to **gain insight**
  - Still, a bit of discipline:
    - Clearly identify assumptions up front
    - Then estimate cost in a principled way

- Foundation for query optimization
  - Can’t choose the fastest scheme without an estimate of speed!

# Cost Model for Analysis

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** (Average) time to read or write disk block
- *Average-case* analyses for *uniform random* workloads
- We will ignore:
  - Sequential vs. Random I/O
  - Pre-fetching
  - Any in-memory costs

➢ ***Good enough to show the overall trends!***

# More Assumptions

- Single record insert and delete.
- Equality selection - exactly one match
- For Heap Files:
  - Insert always appends to end of file.
- For Sorted Files:
  - Files compacted after deletions.
  - Sorted according to search key.

- Question all these assumptions and rework
  - As an exercise to study for tests, generate ideas

# Heap Files & Sorted Files

**B:** The number of data pages = 5
**R:** Number of records per page = 2
**D:** (Average) time to read or write disk page = 5 ms

## Heap File

| | | | | |
|---|---|---|---|---|
| 2, 5 | 1, 6 | 4, 7 | 3, 10 | 8, 9 |

## Sorted File

| | | | | |
|---|---|---|---|---|
| 1, 2 | 3, 4 | 5, 6 | 7, 8 | 9, 10 |

Records are just integers

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | | |
| **Equality Search** | | |
| **Range Search** | | |
| **Insert** | | |
| **Delete** | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | | |
| **Equality Search** | | |
| **Range Search** | | |
| **Insert** | | |
| **Delete** | | |

# Scan all the Records?


Heap File

| | | | | |
|---|---|---|---|---|
| 2, 5 | 1, 6 | 4, 7 | 3, 10 | 8, 9 |

Sorted File

| | | | | |
|---|---|---|---|---|
| 1, 2 | 3, 4 | 5, 6 | 7, 8 | 9, 10 |

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Pages touched: ?

Time to read the record: ?

http://hyperboleandahalf.blogspot.com

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | | |
| **Range Search** | | |
| **Insert** | | |
| **Delete** | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | | |
| **Range Search** | | |
| **Insert** | | |
| **Delete** | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | | |
| **Range Search** | | |
| **Insert** | | |
| **Delete** | | |

# Find Key 8

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Heap File


Pages touched **on average**?

- **P**(i): Probability of key on page **i** is: **1/B**
- **T**(i): # Pages touched if key on page **i** is: **i**
- Therefore the expected # pages touched:

$$\sum_{i=1}^{B} \mathrm{T}(i)\mathbf{P}(i) = \sum_{i=1}^{B} i\frac{1}{B} = \frac{B(B+1)}{2B} \approx \frac{B}{2}$$

# Find Key 8

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Heap File


Pages touched **on average**: B/2

Breaking an Assumption:

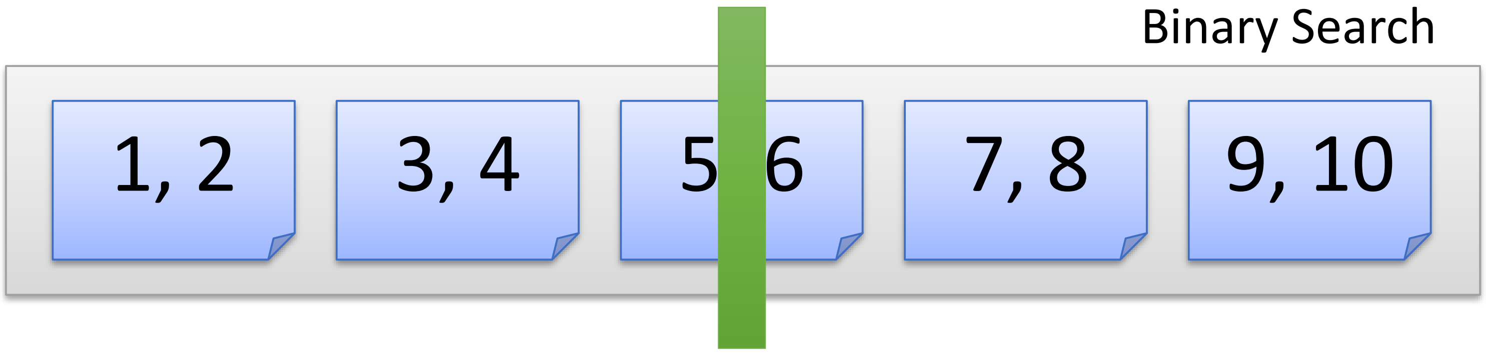
*What if there was more than one key?*

- Need to check all pages → B

# Find Key 8

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Sorted File



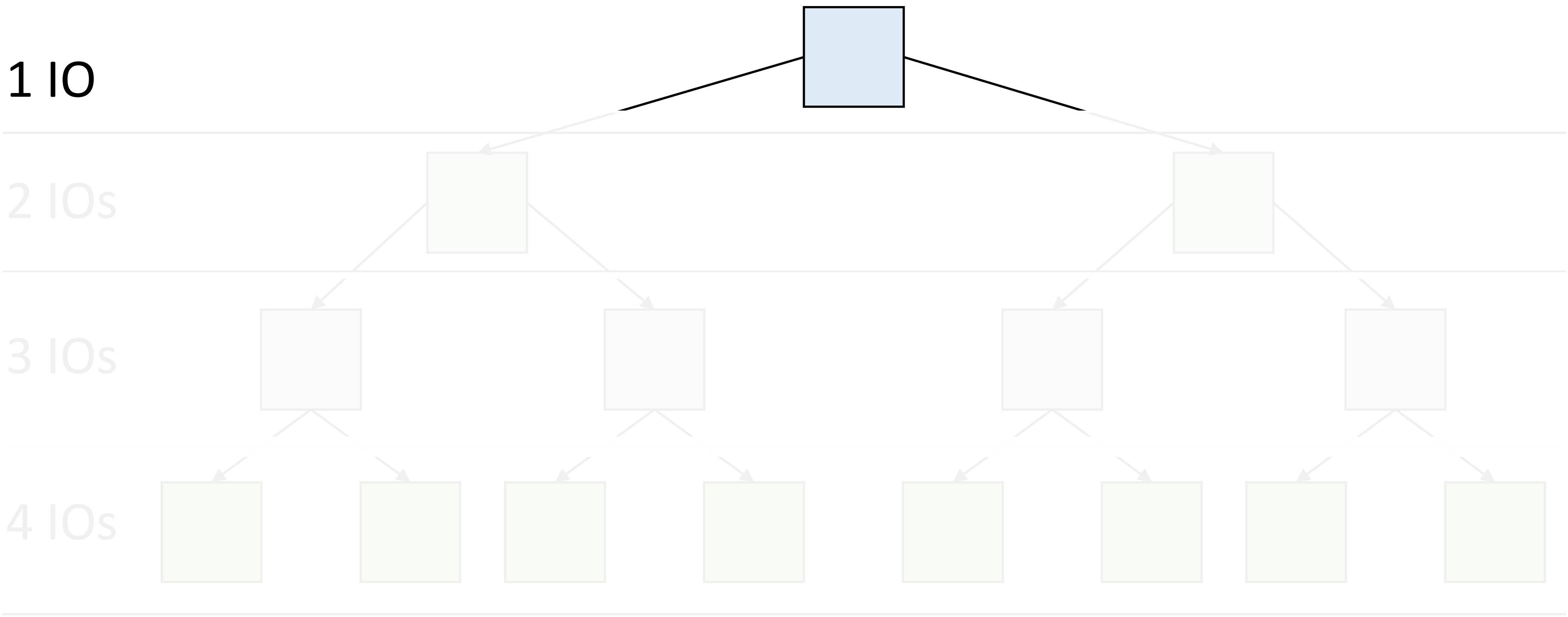

**Worst-case:** Pages touched in binary search

- $\mathbf{Log_2 B}$

**Average-case:** Pages touched in binary search

- $\mathbf{Log_2 B}$**?**

# Average Case Binary Search




Expected Number of Reads: 1 (1 / B) + 2 ( 2 / B) + 3 (4 / B) + 4 (8 / B)

Average ≈ Worst

$$\sum_{i=1}^{\log_2 B} i\frac{2^{i-1}}{B} = \frac{1}{B}\sum_{i=1}^{\log_2 B} i2^{i-1} = \boxed{\log_2 B} - \frac{B-1}{B}$$

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | | |
| **Insert** | | |
| **Delete** | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | | |
| **Insert** | | |
| **Delete** | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | | |
| **Insert** | | |
| **Delete** | | |

# Keys between 7 and 9

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Heap File


Always touch all blocks. Why?

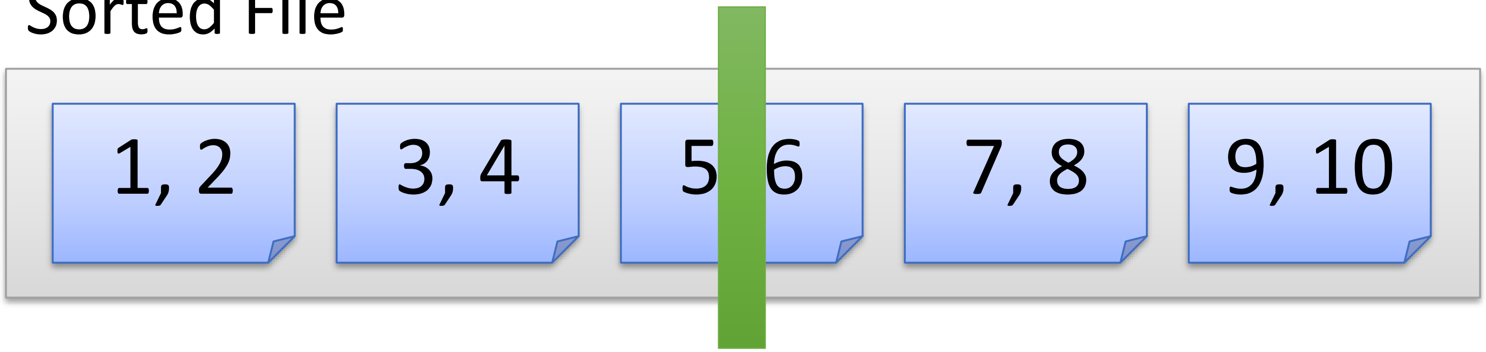
# Keys between 7 and 9

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Heap File


Always touch all blocks. Why?

Sorted File




1. Find beginning of range
2. Scan right

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ |
| **Insert** | | |
| **Delete** | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ |
| **Insert** | | |
| **Delete** | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ |
| **Insert** | | |
| **Delete** | | |

# Insert 4.5

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Heap File


Stick at the end of the file. Cost? ***2D***

Why 2?

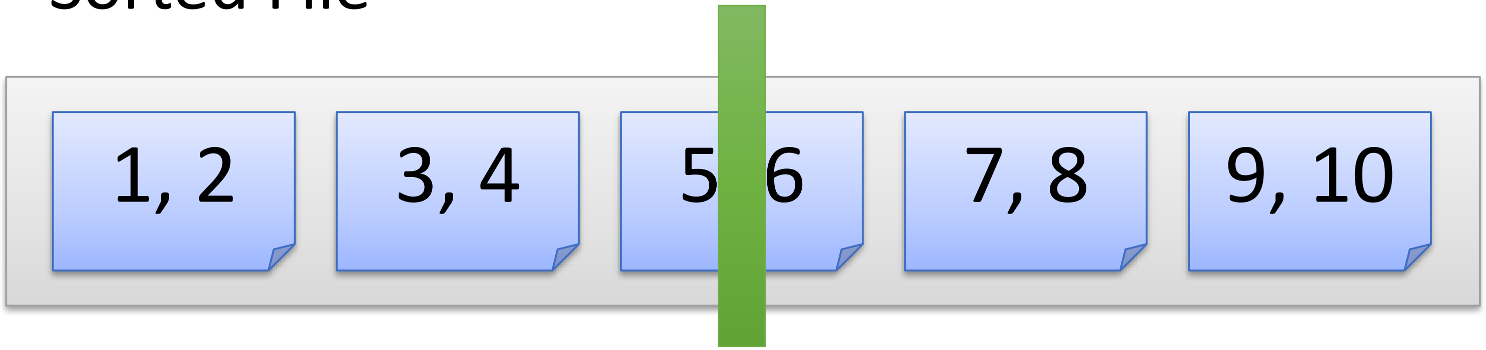
# Insert 4.5

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Heap File


Read last page, append, write. ***2D***

Sorted File




1. Find location for record: $\mathbf{Log_2 B}$

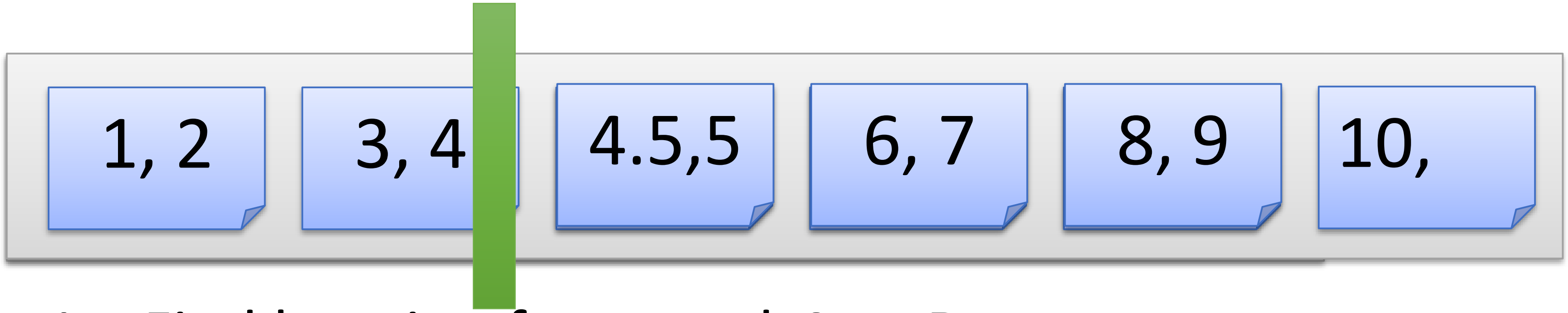
# Insert 4.5

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Heap File


Read last page, append, write. ***2D***

Sorted File




1. Find location for record: $\mathbf{Log_2 B}$
2. Insert and shift rest of file. Cost? **2 (B/2)** **Why?**

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ |
| **Insert** | 2D | $((\log_2 B)+B)D$ |
| **Delete** | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ |
| **Insert** | 2D | $((\log_2 B)+B)D$ |
| **Delete** | | |

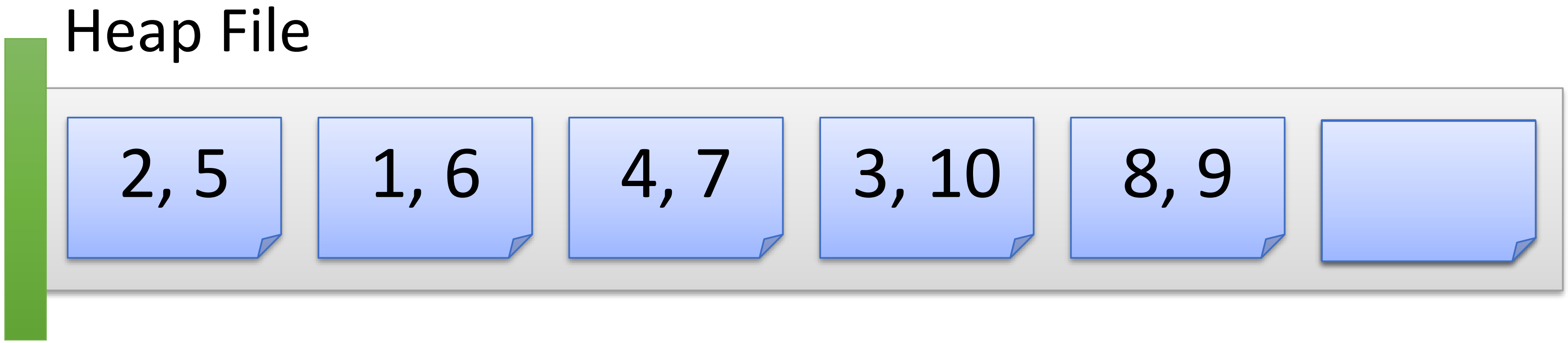
# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ |
| **Insert** | 2D | $((\log_2 B)+B)D$ |
| **Delete** | | |

# Delete 4.5

**B:** data pages
**R:** records per page
**D:** Avg. IO Time




Average case to find record: **B/2 Reads**

Delete record from page.

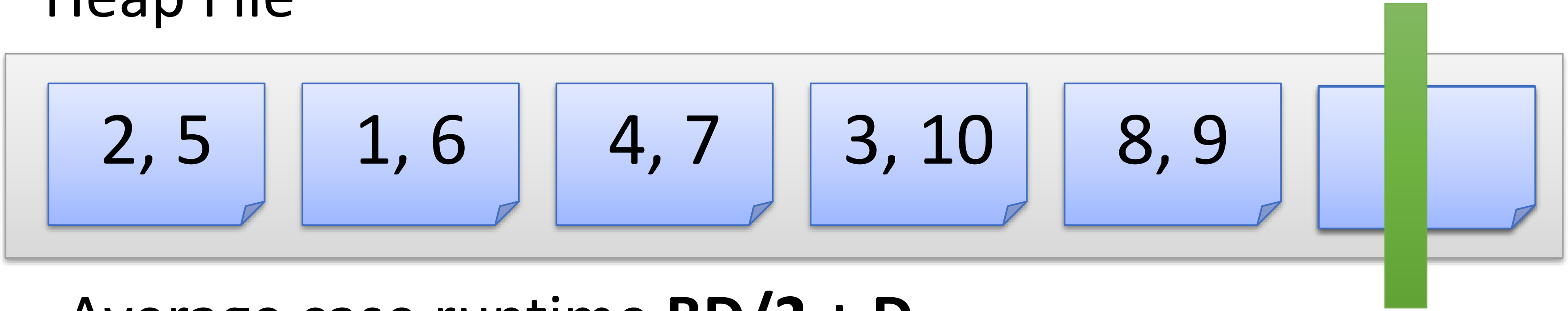
**Cost?** **(B/2 + 1)D**

**Why + 1?**

# Delete 4.5

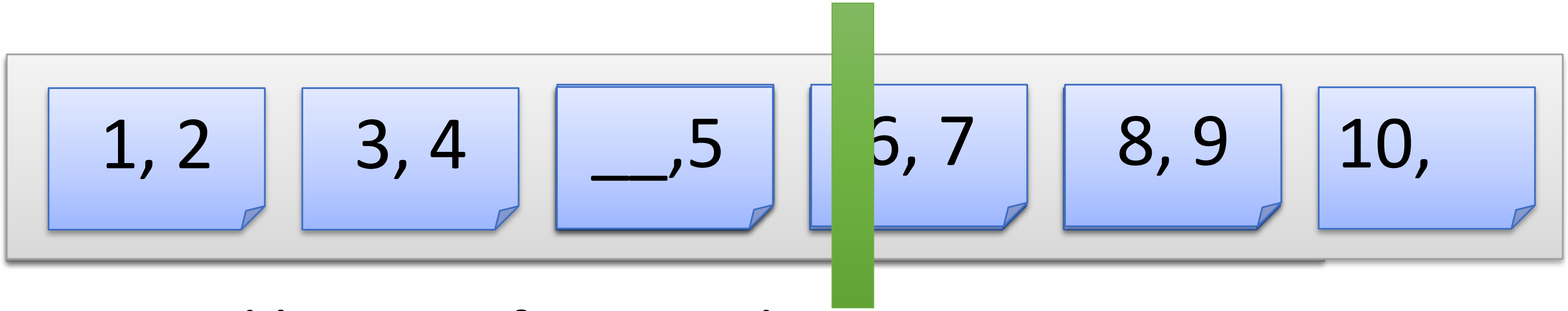
**B:** data pages
**R:** records per page
**D:** Avg. IO Time

Heap File




Average case runtime **BD/2 + D**

Sorted File

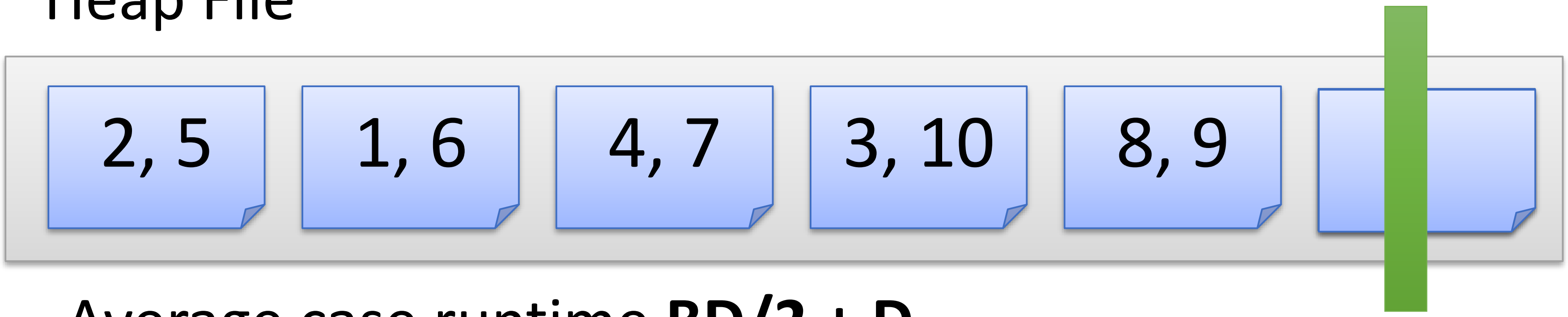



1. Find location for record: $\mathbf{Log_2 B}$
2. Delete record in page → Gap!

**B:** data pages
**R:** records per page
**D:** Avg. IO Time

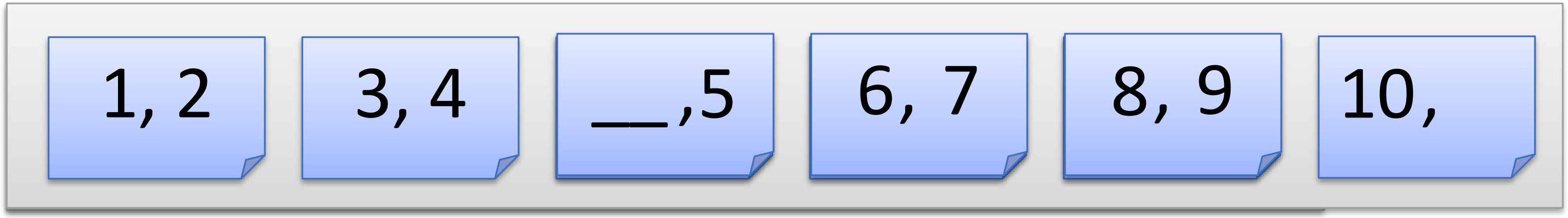
# Delete 4.5

Heap File




Average case runtime **BD/2 + D**

Sorted File




1. Find location for record: **$\mathbf{Log_2B}$**
2. Shift rest of file left by 1 record: **2 (B/2)**

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File |
|---|---|---|
| **Scan all records** | BD | BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ |
| **Insert** | 2D | $((\log_2 B)+B)D$ |
| **Delete** | 0.5BD + D | $((\log_2 B)+B)D$ |

**Which is better?**

# 2. Indexes: Motivations & Basics

“If you don’t find it in the index, look very carefully through the entire catalog”

- Sears, Roebuck and Co., Consumers Guide, 1897

# Index Motivation

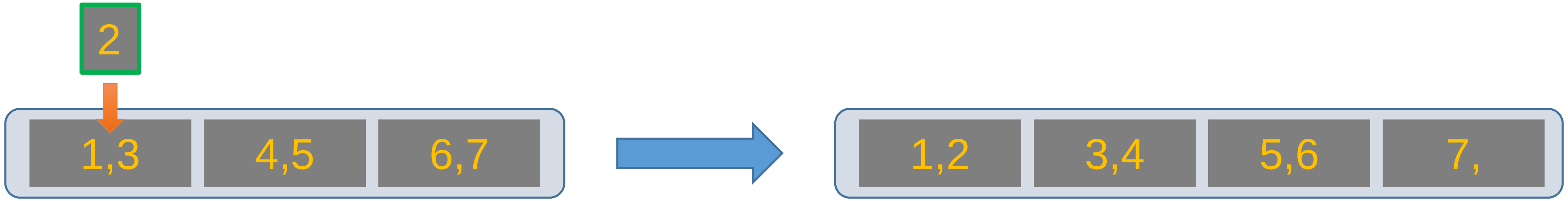
Person(name, age)

- Suppose we want to search for people of a specific age
- ***First idea:*** Sort the records by age… we know how to do this fast!
- How many IO operations to search over ***N=B*R sorted*** records?
  - Simple scan: ***O(BD)***
  - Binary search: ***O($\log_2 B$*D)***

Could we get even cheaper search? E.g. go from $\log_2 B$ → $\log_{200} B$?

# Index Motivation

- What about if we want to **insert** a new person, but keep the list sorted?




- We would have to potentially shift ***N*** records, requiring up to **~ 2*B** IO operations!
  - We could leave some "slack" in the pages…

Could we get faster insertions?

# Index Motivation

- What about if we want to be able to search quickly along multiple attributes (e.g. not just age)?
  - We could keep multiple copies of the records, each sorted by one attribute set… this would take a lot of space

Can we get fast search over multiple attribute (sets) without taking too much space?

We'll create separate data structures called *indexes* to address all these points

# Further Motivation for Indexes: NoSQL!

- NoSQL engines are (basically) ***just indexes!***
  - A lot more is left to the user in NoSQL… one of the primary remaining functions of the DBMS is still to provide index over the data records, for the reasons we just saw!
  - Sometimes use B+ Trees (covered next), sometimes hash indexes (not covered here)

Indexes are critical across all DBMS types

# Indexes: High-level

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Search key properties
    - Any subset of fields
    - is **not** the same as *key of a relation*

- *Example:*

Product(name, maker, price)

On which attributes would you build indexes?

# More precisely

- An *index* is a **data structure** mapping search keys to sets of rows in a database table

  - Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table

- An index can store the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)

# Conceptual Example

What if we want to return all books published after 1867? The above table might be very expensive to search over row-by-row…

**Russian_Novels**

| BID | Title | Author | Published | Full_text |
|---|---|---|---|---|
| 001 | *War and Peace* | Tolstoy | 1869 | … |
| 002 | *Crime and Punishment* | Dostoyevsky | 1866 | … |
| 003 | *Anna Karenina* | Tolstoy | 1877 | … |

```
SELECT *
FROM Russian_Novels
WHERE Published > 1867
```

# Conceptual Example

**By_Yr_Index**

| Published | BID |
|---|---|
| 1866 | 002 |
| ***1869*** | ***001*** |
| ***1877*** | ***003*** |

**Russian_Novels**

| BID | Title | Author | Published | Full_text |
|---|---|---|---|---|
| 001 | *War and Peace* | Tolstoy | 1869 | … |
| 002 | *Crime and Punishment* | Dostoyevsky | 1866 | … |
| 003 | *Anna Karenina* | Tolstoy | 1877 | … |

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

# Conceptual Example

## By_Yr_Index

| Published | BID |
| --- | --- |
| 1866 | 002 |
| 1869 | 001 |
| 1877 | 003 |

## By_Author_Title_Index

| Author | Title | BID |
| --- | --- | --- |
| Dostoyevsky | Crime and Punishment | 002 |
| Tolstoy | Anna Karenina | 003 |
| Tolstoy | War and Peace | 001 |

## Russian_Novels

| BID | Title | Author | Published | Full_text |
| --- | --- | --- | --- | --- |
| 001 | *War and Peace* | Tolstoy | 1869 | … |
| 002 | *Crime and Punishment* | Dostoyevsky | 1866 | … |
| 003 | *Anna Karenina* | Tolstoy | 1877 | … |

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures…

# Covering Indexes

**By_Yr_Index**

| Published | BID |
|---|---|
| 1866 | 002 |
| 1869 | 001 |
| 1877 | 003 |

We say that an index is **covering** *for a specific query* if the index contains all the needed attributes- ***meaning the query can be answered using the index alone!***

The "needed" attributes are the union of those in the SELECT and WHERE clauses…

Example:

```
SELECT Published, BID
FROM Russian_Novels
WHERE Published > 1867
```

# Operations on an Index

- <u>Search</u>: Quickly find all records which meet some *condition on the search key attributes*
  - More sophisticated variants as well. Why?

- <u>Insert / Remove</u> entries
  - Bulk Load / Delete. Why?

Indexing is one the most important features provided by a database for performance

# Kinds of Lookups Supported?

- Basic Selection: <key> <op> <constant>
  - Equality selections (op is =)?
  - Range selections (op is one of <, >, <=, >=, BETWEEN)

- More exotic selections:
  - 2-dimensional ranges ("east of Berkeley and west of Truckee and North of Fresno and South of Eureka")
  - 2-dimensional radii ("within 2 miles of Soda Hall")
  - Common **n-dimensional index**: *R-tree, KD-Tree*
    - Beware of the ***curse of dimensionality***
  - Ranking queries ("10 restaurants closest to Berkeley")
  - Regular expression matches, genome string matches, etc.
  - See [http://en.wikipedia.org/wiki/GiST](http://en.wikipedia.org/wiki/GiST) for more

# Search Key: *Any* Subset of Columns

- Search key needn't be a key of the relation
  - Recall: key of a relation must be unique (e.g., SSN)
  - Search keys don't have to be unique
- **Composite Keys**: more than one column
  - Think: Phone Book <Last Name, First>
  - **Lexicographic order**
  - <Age, Salary>:
    - ✓ Age = 31 & Sal = 400
    - ✓ Age = 55 & Sal > 200
    - ✗ Age > 31 & Sal = 400
    - ✓ Age = 31
    - ✓ Age > 31
    - ✗ Sal = 300

| SSN | Last Name | First Name | Age | Salary |
|---|---|---|---|---|
| 123 | Adams | Elmo | 31 | $400 |
| 443 | Grouch | Oscar | 32 | $300 |
| 244 | Oz | Bert | 55 | $140 |
| 134 | Sanders | Ernie | 55 | $400 |

✗ Means that the index is unable to exclude all entries that are not in the result set.

# Data Entries: How are they stored?

- What is the representation of data in the index?
  - Actual data or pointer(s) to the data
- How is the data stored in the data file?
  - Clustered or unclusted with respect to the indexed
- Big Impact on Performance

# How is data stored in the index

- Three alternatives:
  1. **By Value:** actual data record (with key value **k**)
  2. **By Reference:** <**k**, rid of matching data record>
  3. **By List of Refs.:** <**k**, list of rids of *all* matching data records>

- Choice is orthogonal to the indexing technique.
  - B+ trees, hash-based structures, R trees, GiSTs, …

- Can have multiple (different) indexes per file.
  - E.g. file sorted by *age*, with a hash index on *salary* and a B+tree index on *name*.

# Alternatives for Data Entries (Contd.)

Alternative 1 (By Value):
Actual data record (with key value ***k***)

- Index as a file organization for records
  - Alongside heap files or sorted files
- No "**pointer lookups**" to get data records
  - Following record ids
- Could a single relation have multiple indexes of this form?
  - Probably but it would be a bad idea. Why?

# Alternatives for Data Entries (Contd.)

Alternative 2 (By Reference)

<**k**, rid of matching data record>

and Alternative 3 (By List of References)

<**k**, list of rids of matching data records>

Alternative 2

| Key | Record Id |
|---|---|
| Gonzalez | 1 |
| Gonzalez | 2 |
| Gonzalez | 3 |
| Hong | 4 |

| SSN | Last Name | First Name | Salary |
|---|---|---|---|
| 123 | Gonzalez | Amanda | $400 |
| 443 | Gonzalez | Joey | $300 |
| 244 | Gonzalez | Jose | $140 |
| 134 | Hong | Sue | $400 |

Alternative 3

| Key | Record Id |
|---|---|
| Gonzalez | {1,2,3} |
| Hong | 4 |

- Alts. 2 or 3 needed to support multiple indexes per relation!
- Alt. 3 more compact than Alt. 2
- For very large rid lists, single data entry spans multiple blocks

# Clustered vs. Unclustered Index

- In a clustered index:
  - index data entries are stored in (approximate) order by value of search keys in data records

# Clustered vs. Unclustered Index

- In a clustered index:
  - index data entries are stored in (approximate) order by value of search keys in data records

Clustered

| Key | Record Id |
|---|---|
| Gonzalez | 1 |
| Gonzalez | 2 |
| Gonzalez | 3 |
| Hong | 4 |

| SSN | Last Name | First Name | Salary |
|---|---|---|---|
| 123 | Gonzalez | Amanda | $400 |
| 443 | Gonzalez | Joey | $300 |
| 244 | Gonzalez | Jose | $140 |
| 134 | Hong | Sue | $400 |

Unclustered

| Key | Record Id |
|---|---|
| Gonzalez | 3 |
| Hong | 4 |
| Gonzalez | 1 |
| Gonzalez | 2 |

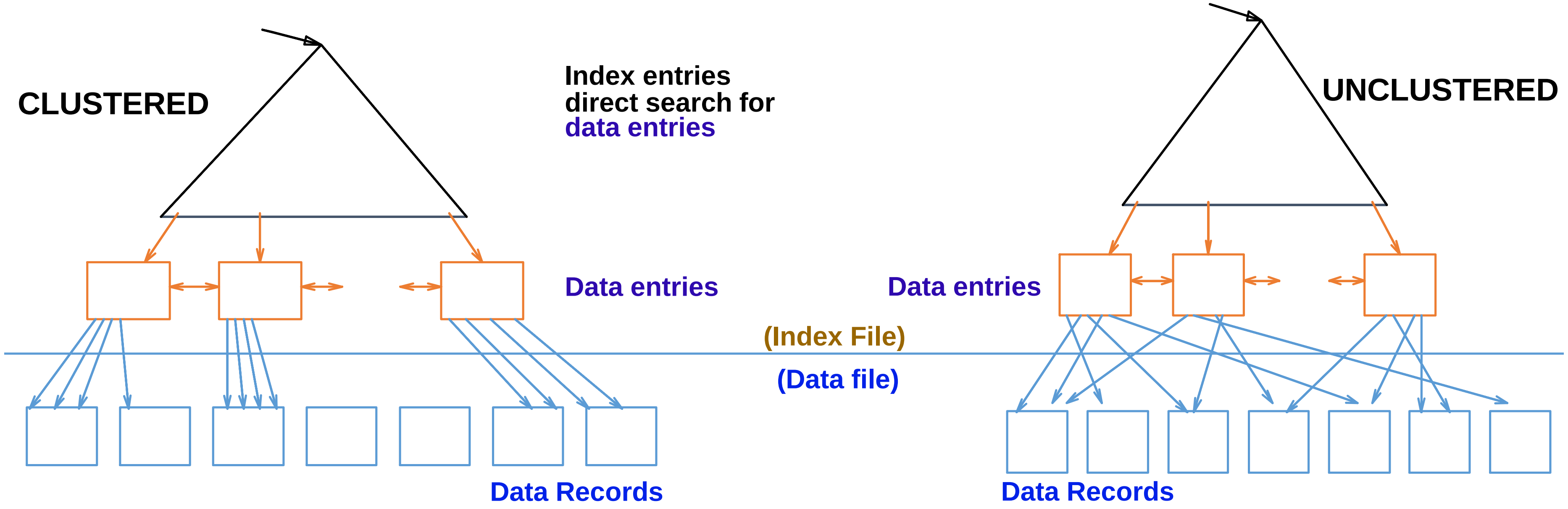
# Clustered vs. Unclustered Index

- In a clustered index:
  - index data entries are stored in (approximate) order by value of search keys in data records
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
  - Can Alternative 1 be un-clustered?
    - No, but clustered does not imply alt. 1 (earlier figure)
- Note: there is another definition of "clustering"
  - **Data Mining/AI**: grouping similar items in n-space

# Clustered vs. Unclustered Index

Alternative 2 (by ref) data entries, data records in a Heap file.

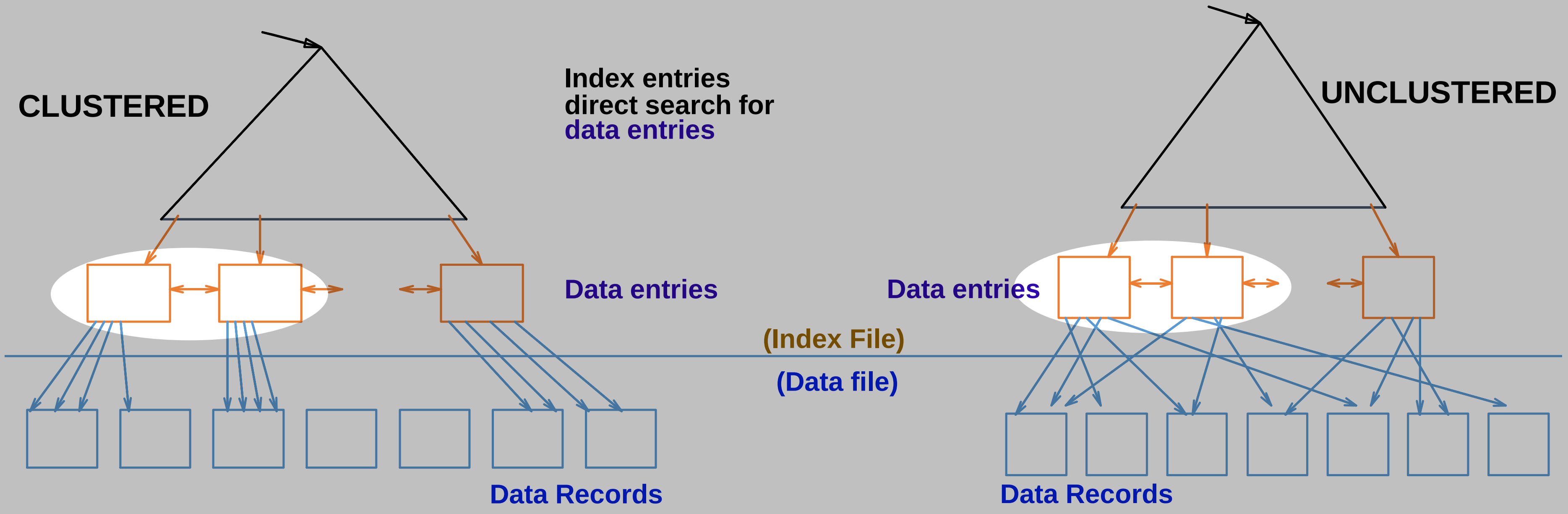
- To build clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
  - Thus, order of data recs is “close to”, but not identical to, the sort order.



# Clustered vs. Unclustered Index

Alternative 2 (by ref) data entries, data records in a Heap file.

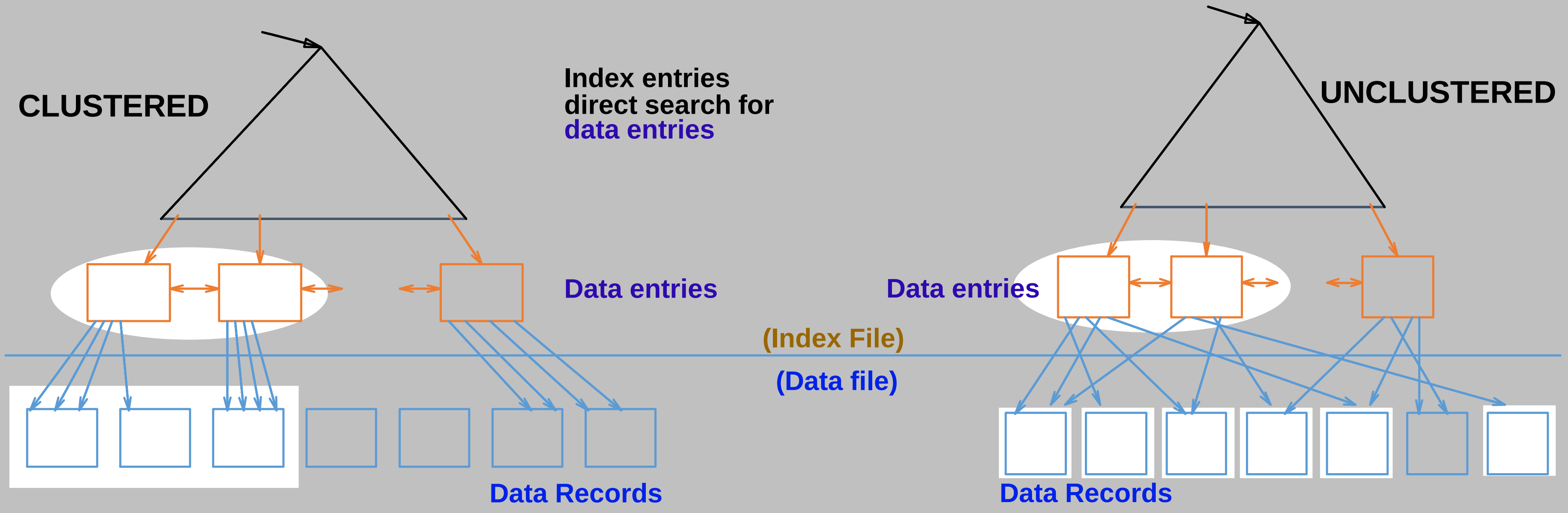
- To build clustered index, first sort the Heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
  - Thus, order of data recs is “close to”, but not identical to, the sort order.



# Clustered vs. Unclustered Index

Alternative 2 (by ref) data entries, data records in a Heap file.

- To build clustered index, first sort the Heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
  - Thus, order of data recs is “close to”, but not identical to, the sort order.



# Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**

- For exact search, no difference between clustered / unclustered

- For range search over N=B*R values: difference between **1 random IO + B sequential IO**, and **B random IO**:
  - A random IO costs ~ 10ms (sequential much much faster)
  - For B = 100,000 - **difference between ~10ms and ~17min!**

# Unclustered vs. Clustered Indexes

- Clustered Pros
  - Efficient for range searches
  - Potential locality benefits?
    - Sequential disk access, prefetching, etc.
  - Support certain types of compression
    - More soon on this topic

- Clustered Cons
  - More expensive to maintain
    - Need to update index data structure
    - Solution: on the fly or "lazily" via reorgs
  - Heap file usually only **packed to 2/3** to accommodate inserts

# High-level Categories of Index Types

- B-Trees *(covered next)*
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will look at a variant called **B+ Trees***

- Hash Tables *(not covered)*
  - There are variants of this basic structure to deal with IO
  - Called ***linear*** or ***extendible hashing-*** IO aware!

The data structures we present here are "IO aware"

**Real difference between structures**: costs of ops *determines which index you pick and why*