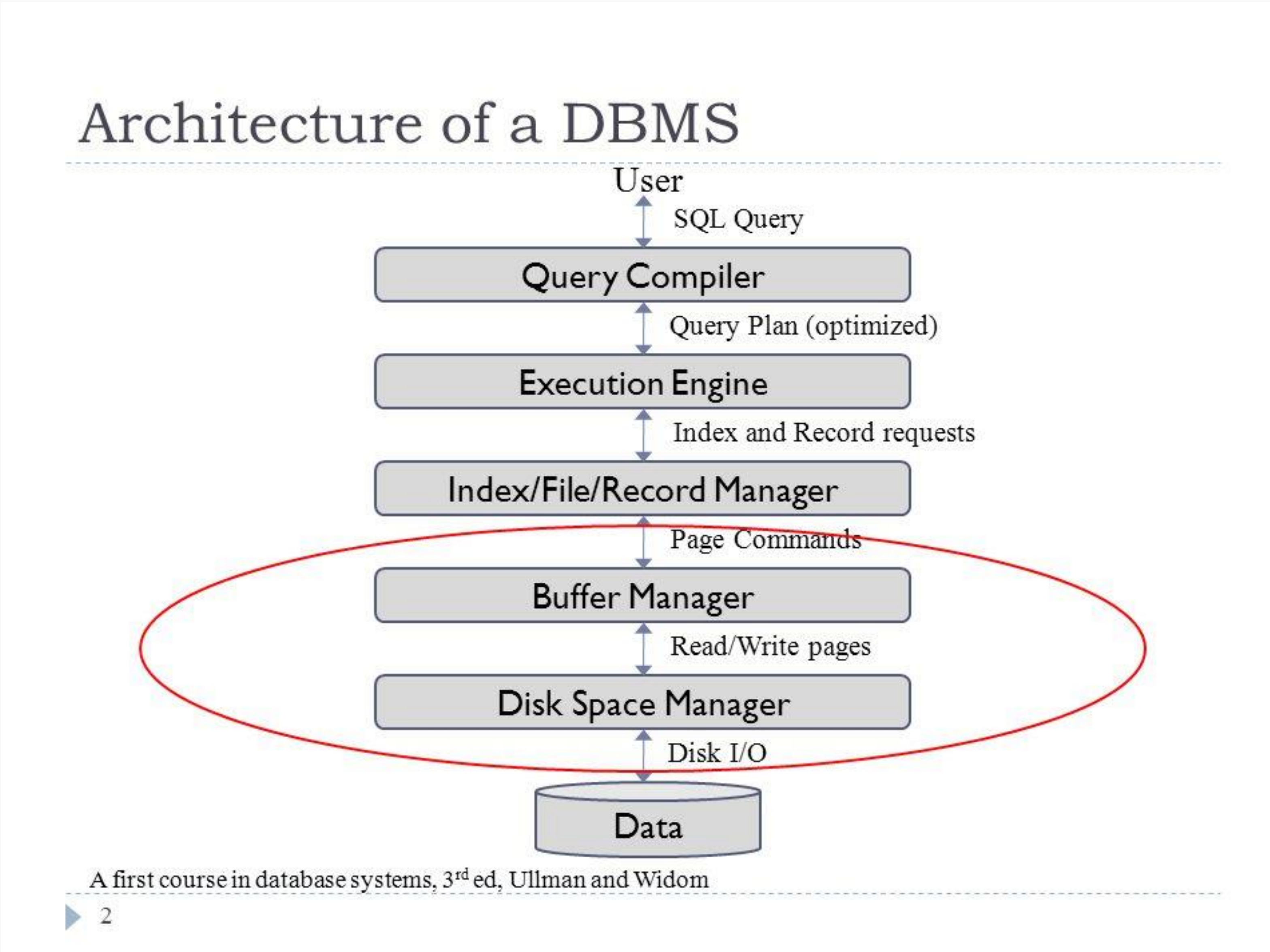
Storage and Indexing

Motivation

- DBMS stores vast quantities of data
- Data is stored on external storage devices and fetched into main memory as needed for processing
- Page is unit of information read from or written to disk. (in DBMS, a page may have size 8KB or more).
- Data on external storage devices :
  - Disks: Can retrieve random page at fixed cost (I/O operations).
    But reading several consecutive pages is much cheaper (i.e. faster) than reading them in random order
  - Tapes: Can only read pages in sequence.
    Cheaper than disks; used for archival storage.

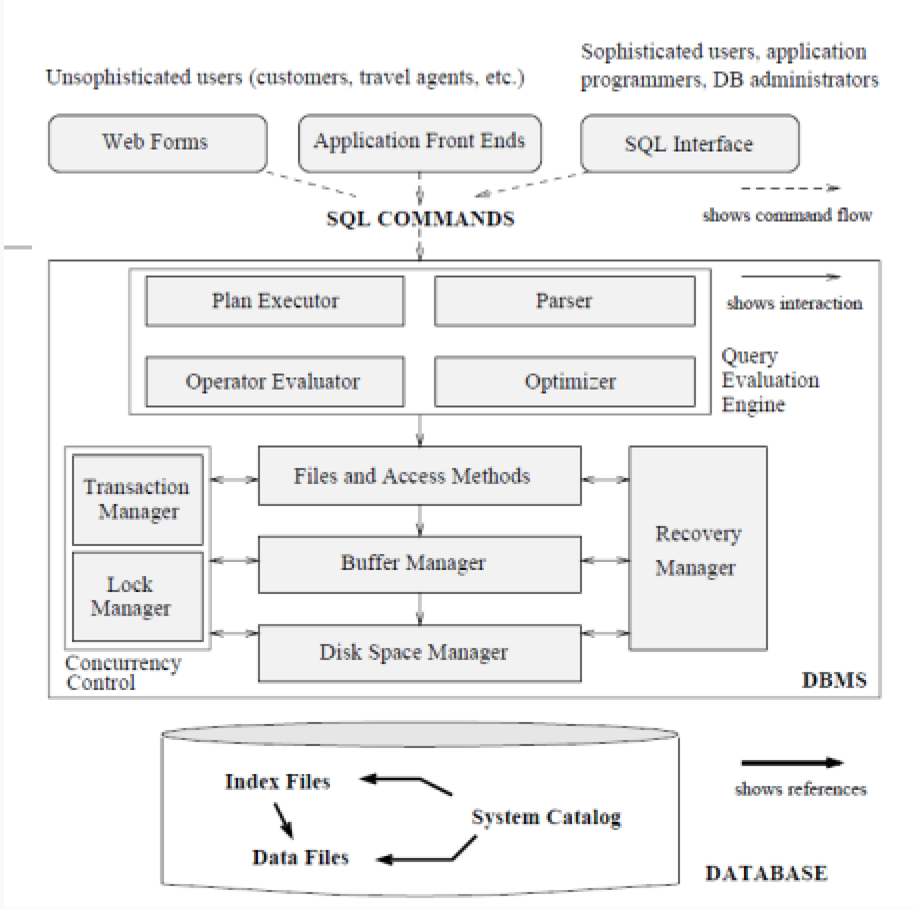
- Cost of page I/O dominates cost of typical database operations

Data on External Storage



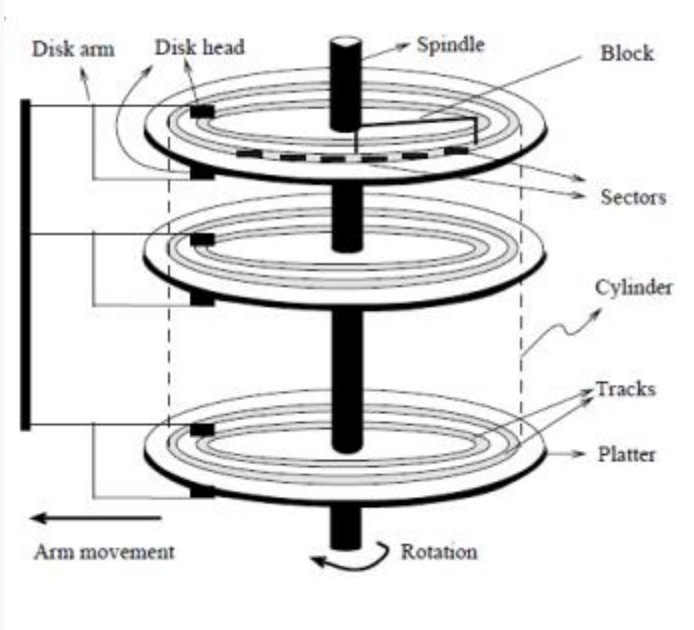
introduction

- **DBMS** abstracts data as a collection of **records** stored in a **file**.
- A file is a set of **pages**, each contain certain **set of records**.
- The **files layer** is responsible or data organization for fast data retrieval.
- **File organization**: a way of organizing records in a file.
- Each file organization makes certain operations efficient, but other operations expensive



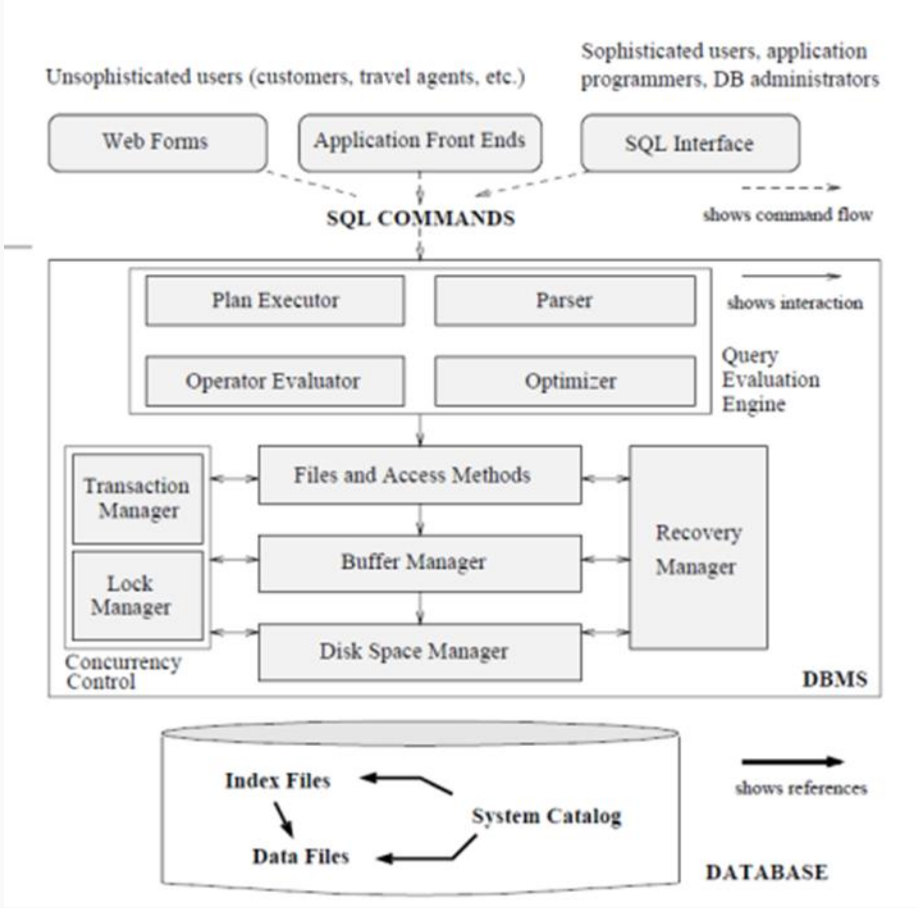
Data on External Storage

- **Hard disks** are the primary storage devices for DBMS
- The **taps** are used for archiving.
- The unit of information read from or written from disk is a **page**.
- A page is **typically 4KB** or **8KB**
- The cost of page I/O is the **most expensive** operation.
- Disks have **fixed cost per page**.
- Each record in a file has a unique identifier called **rid**.
- Using the **rid**, we can identify the **page address**



Data on External Storage

- The **buffer manager** is responsible for loading a page into memory.
- When the files layer wants to access a certain page, it asks the **buffer manager** to load it into memory (if it is not already there)
- Space on disk is managed by **disk space manager.**



Data on External Storage

- File organization:
  - Method of arranging a file of records on external storage.
  - Record id (rid) is sufficient to physically locate record
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields
- Architecture: Buffer manager stages pages from external storage to main memory buffer pool.

Multiple File Organizations

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

- **Heap Files:**
  - is the simplest file organization: records are stored randomly across the pages.
  - Suitable when typical access is a full scan of all records
  - Unordered collection of records
  - Add/Remove records: Easy (Cost?)
- **Sorted Files:**
  - Best for retrieval in search key order, or a range of records is needed
  - Arrange and store collection of records in sorted manner.
  - Add/Remove records: Easy or not (Cost?)
- **Clustered Files & Indexes:** Group data into block to enable fast lookup and efficient modifications. (More on this soon …)
  An **index** is a data structure that allows fast retrieval of data records.
  We can create several indexes for same data file, each with different search key.

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 Today

- Big picture overheads for data access
    - Then estimate cost in a principled way

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

Cost Model & Analysis

Cost Model for Analysis

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** (Average) time to read/write disk block

- Average case analysis 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

Heap Files & Sorted Files

Heap File

Sorted File

- **B:** The number of data blocks = 5
- **R:** Number of records per block = 2
- **D:** (Average) time to read/write disk block = 5ms

Cost of Operations

	Heap File	Sorted File
Scan all records		
Equality Search		
Range Search		
Insert		
Delete		

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations

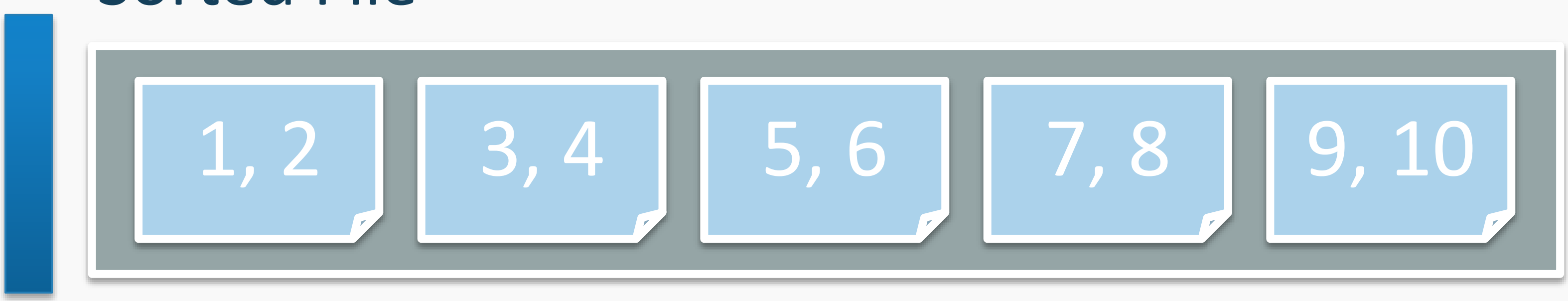
	Heap File	Sorted File
Scan all records		
Equality Search		
Range Search		
Insert		
Delete		

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Scan All Records

Heap File

Sorted File



- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Pages touched: ?

Time to read the record: ?

Cost of Operations

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search		
Range Search		
Insert		
Delete		

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

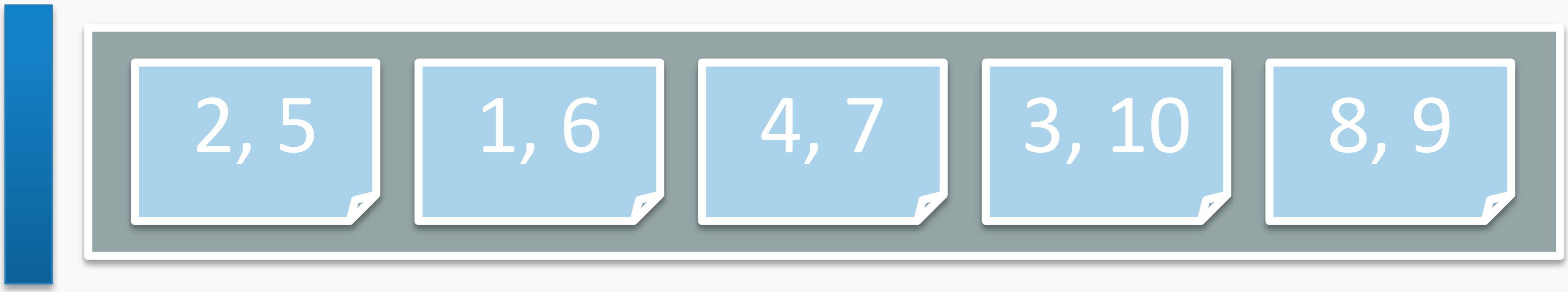
Cost of Operations

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search		
Range Search		
Insert		
Delete		

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Find Key 8

Heap File



Pages touched on average?

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

Heap File

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

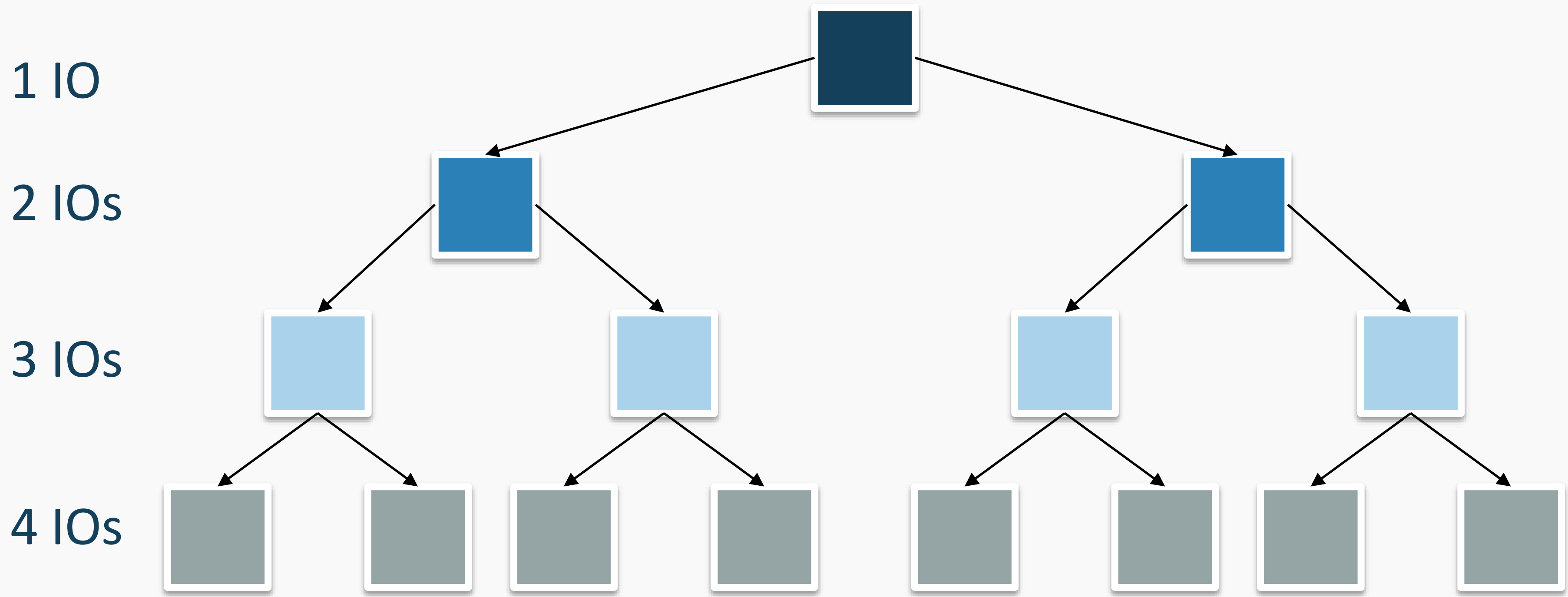
- **Breaking an assumption**
  - What if there was more than one key?
  - Need to check all the pages → B

Find Key 8

Sorted File

- **Worst-case:** Pages touched in binary search
  - **$\log_2 B$**
- **Average-case:** Pages touched in binary search
  - **$\log_2 B$?**

Average Case Binary Search



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

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

Cost of Operations

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

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations

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

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

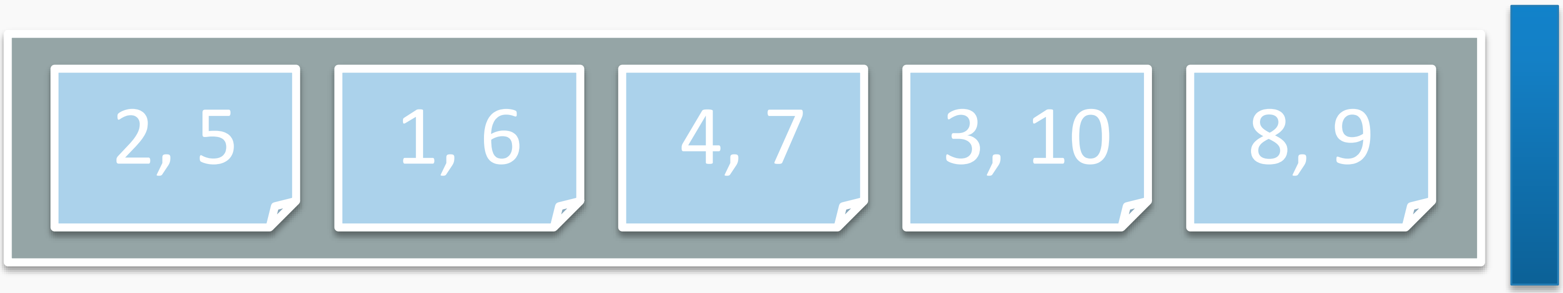
Find Keys Between 7 and 9

Heap File

Always touch all blocks. Why?

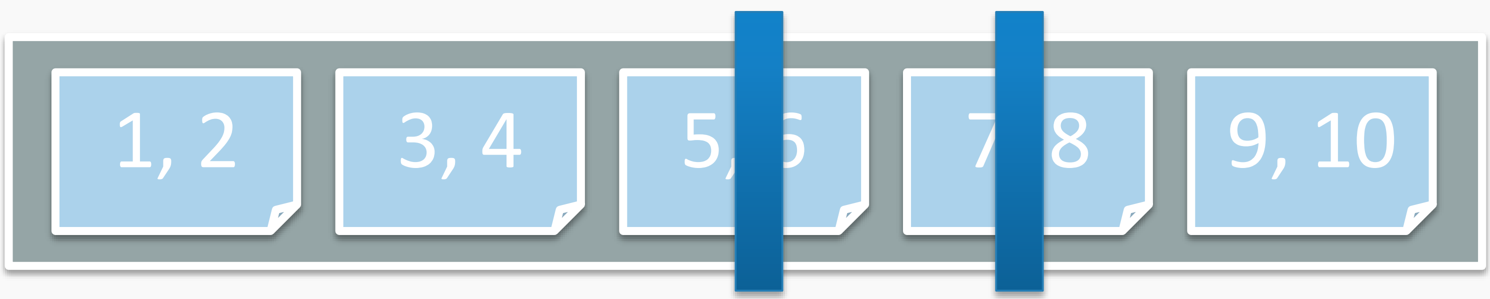
Find Keys Between 7 and 9

Heap File



Always touch all blocks. Why?

Sorted File



- Find beginning of range
- Scan right

Cost of Operations

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

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

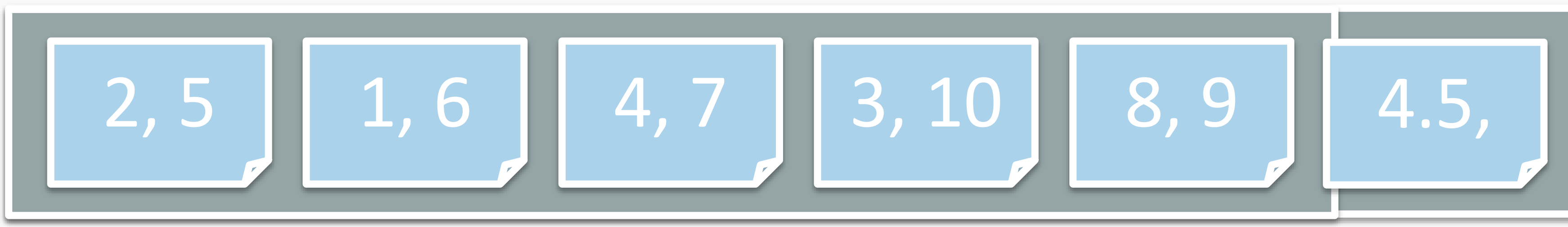
Cost of Operations

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

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Insert 4.5

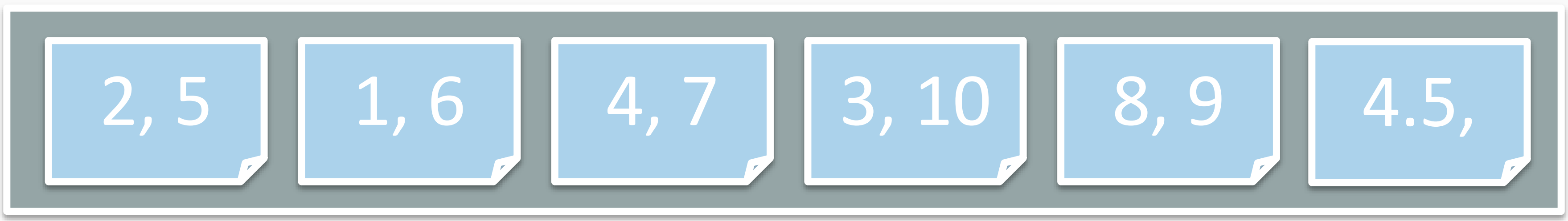
Heap File



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

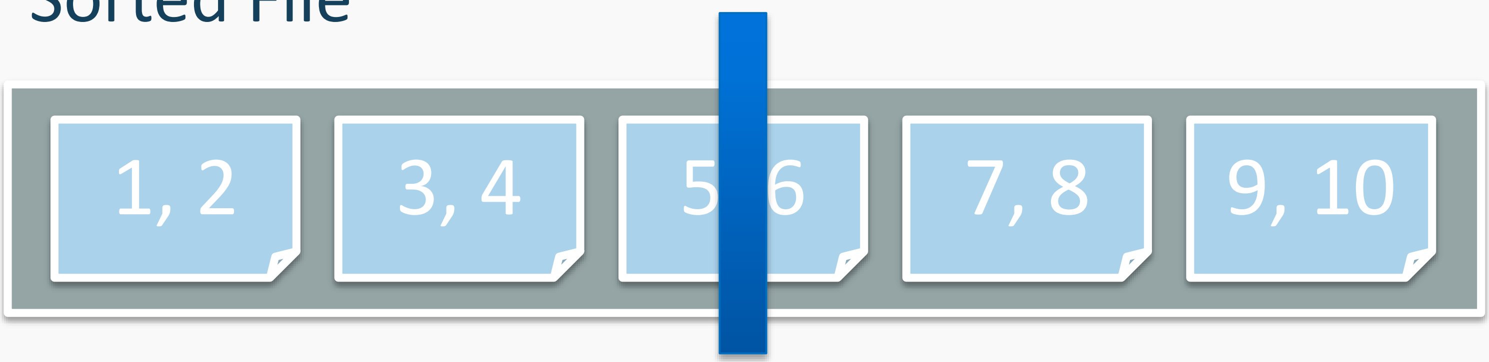
Insert 4.5

Heap File



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

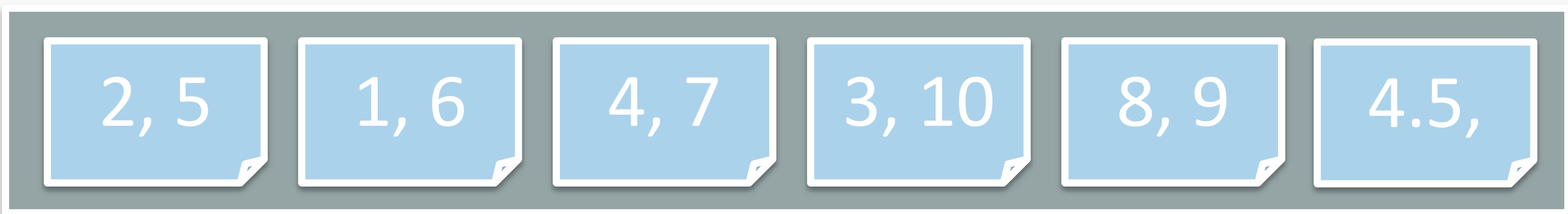
Sorted File



- Find location for record: $\mathbf{log_2 B}$

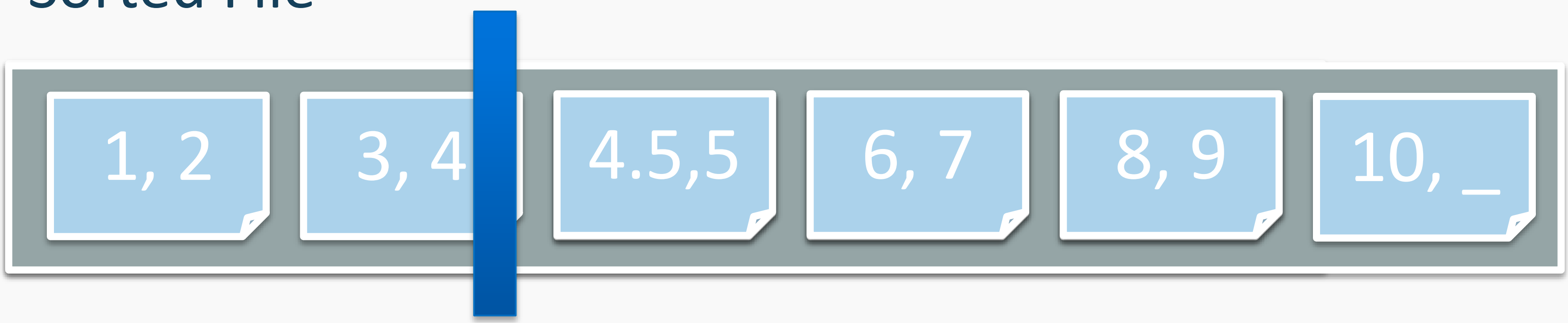
Insert 4.5

Heap File



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

Sorted File



- Find location for record: $\mathbf{log_2 B}$
- Insert and shift rest of file **Cost? 2*B/2 Why?**

Cost of Operations

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search	B*D	$((\log_2 B)$+pages)*D
Insert	2*D	$((\log_2 B)$+B)*D
Delete		

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

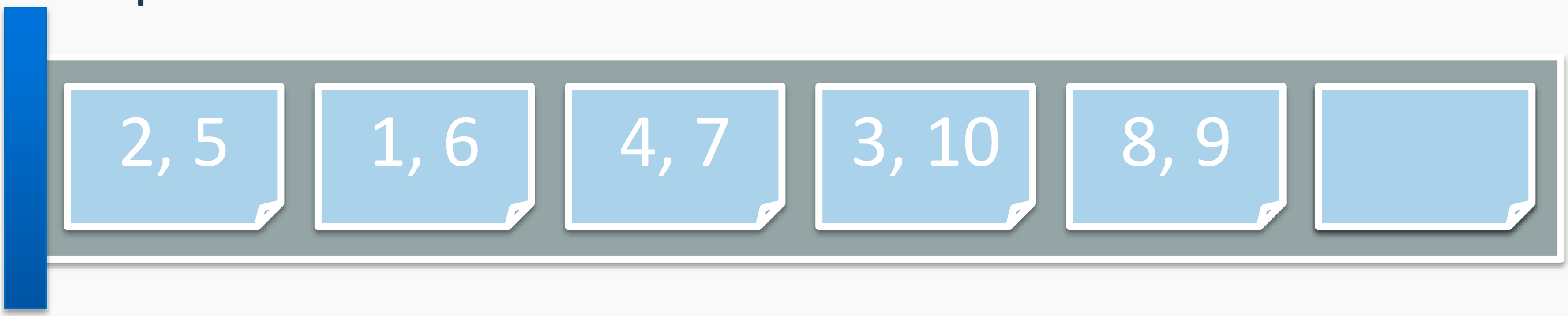
Cost of Operations

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)*D$
Range Search	B*D	$((\log_2 B)+pages)*D$
Insert	2*D	$((\log_2 B)+B)*D$
Delete		

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Delete 4.5

Heap File



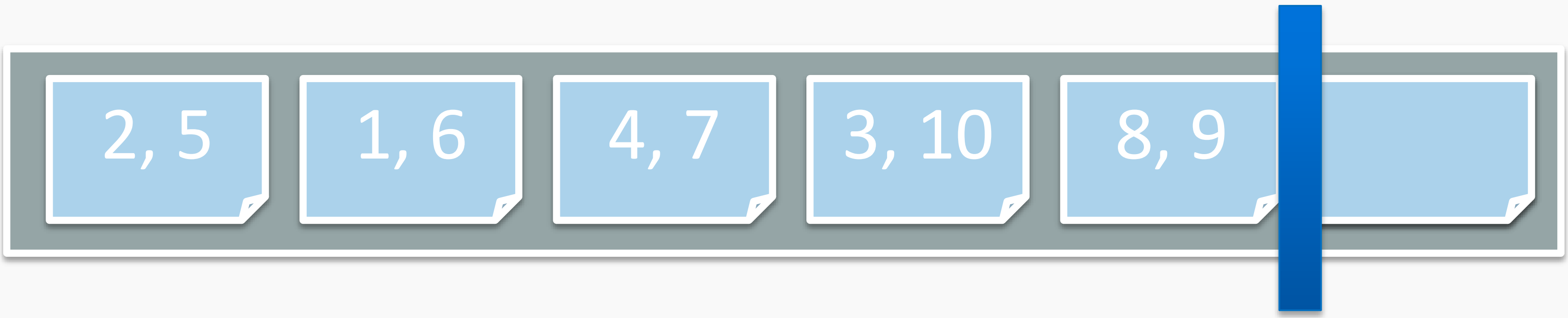
Average case to find the record: **B/2 reads**

Delete record from page

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

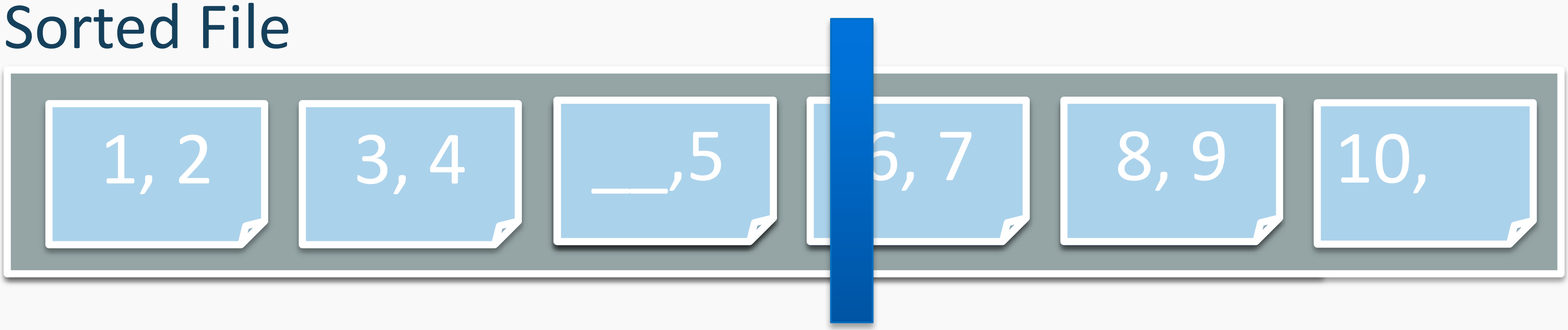
Delete 4.5

Heap File



Average case runtime: **(B/2+1) * D**

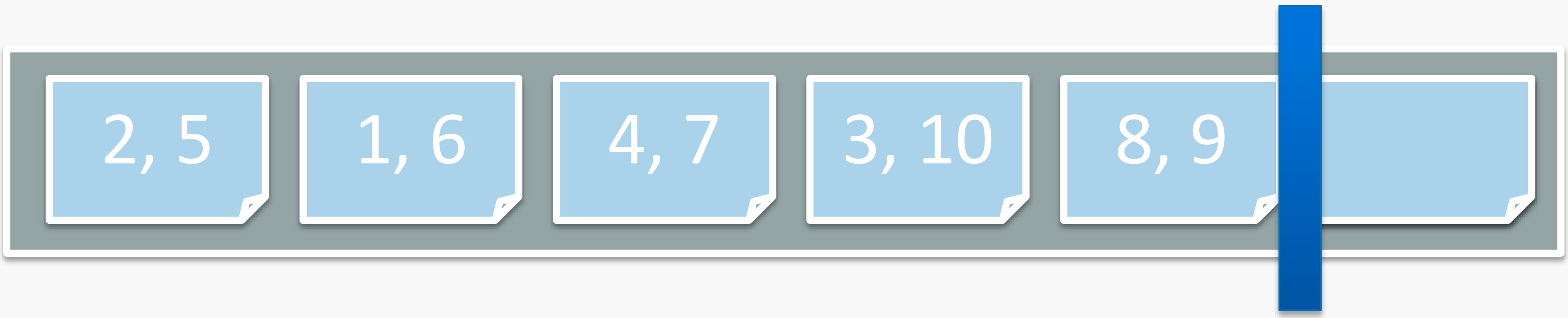
Sorted File



- Find location for record: **$\log_2 B$**
- Delete record in page → Gap

Delete 4.5

Heap File



Average case runtime: **(B/2+1) * D**

Sorted File

- Find location for record: **$\log_2 B$**
- Shift rest of file left by 1 record: **2 * (B/2)**

Cost of Operations

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)*D$
Range Search	B*D	$((\log_2 B)+pages)*D$
Insert	2*D	$((\log_2 B)+B)*D$
Delete	(0.5*B+1)*D	$((\log_2 B)+B)*D$

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Which is better?

Cost of Operations

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search	B*D	$((\log_2 B)$+pages)*D
Insert	2*D	$((\log_2 B)$+B)*D
Delete	(0.5*B+1)*D	$((\log_2 B)$+B)*D

Issues:

- Find
- Range
- Modification

Can we do better?

Indexes

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Indexes

Indexes Overview

- ✓ **Indexing** organizes data records on disk to optimize certain kinds of **retrieval operations**.
- ✓ An **index** is a data structure that enables fast **lookup** of **data entries** by **search key.**
  - **Lookup (retrieval):** may support many different operations
    - Equivalence (i.e. =), range (i.e. >, < , >=), …
  - **Data Entries:** records stored in the index file, (**k,** {items})
    - A data entry with search key value **k**, denoted as **k*.**
    - Could be actual records or record-ids (pointers).
    - We can efficiently search an index to find the desired data entries, and then use these to obtain data records.
  - **Search Key:** any subset of columns (i.e. fields) in the relation.

Search Key: Any **Subset** of Columns?

- **Search key does not require to 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
- Additional indexes can be created on a given collection of data records, each with a different search key,
- **Why indexing used?**
- to speed up search operations that are not efficiently supported by the file organization used to store the data records on disk.

Example

- Consider the Employee Table.
- We can store the records in a file organized as an **index on employee age**;
- which it is an alternative to sorting the file by age (i.e Sorted file).
- Additionally, we can create an auxiliary **index file based on salary**, to speed up queries involving salary.
-

Example: creating different indexes

<age, sal>	rid
19,100	4
20,10	1
20,20	5
24,80	2
25,75	3

<sal,age>	rid
10,20	1
20,20	5
75,25	3
80,24	2
100,19	4

Employee Table

Name	age	sal
Ahmad	20	10
Assad	24	80
Murad	25	75
Moh’d	19	100
Qusai	20	20

<age>	rid
19	4
20	1
20	5
24	2
25	3

<sal>	rid
10	1
20	5
75	3
80	2
100	4

Search Key: Any **Subset** of Columns?

<age, sal>	rid
31,400	1
32,300	2
55,140	3
55,400	4

- 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 & Salary = 400
    - ✓ Age = 55 & Salary > 200
    - ✗ Age > 31 & Salary = 400
    - ✓ Age = 31
    - ✓ Age > 31
    - ✗ Salary = 300

SSN	Name	Age	Salary
123	Ahmad	31	$400
443	Assad	32	$300
244	Moh’d	55	$140
134	Qusai	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 unclustered with respect to the index
- **Big Impact on Performance**

What to store as a data entry in an index?

- Three main alternatives:
  1. **By Value:**

     A data entry **k*** is an actual data record (with search key value **k**).

  2. **By Reference:** <**k**, rid of matching data record>

     A data entry **k*** is a (k, *rid*) pair, where *rid* is the record id of a data record with search key value **k**.

  3. **By List of References:** <**k**, list of rids of *all* matching data records>

     A data entry **k*** is a (k. rid-list) pair, where rid-list is a list of record ids of data records with search key value k.

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

Alternatives for Storing Data Entries

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

- Index as a file organization for records
  - Similar to 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?

Alternatives for Storing Data Entries

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>

By Reference

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

By List of references

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

- Alternatives 2 or 3 needed to support multiple indexes per table!
- Alternative 3 more compact than alternative 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
Gonzalez	1
Hong	4
Gonzalez	2

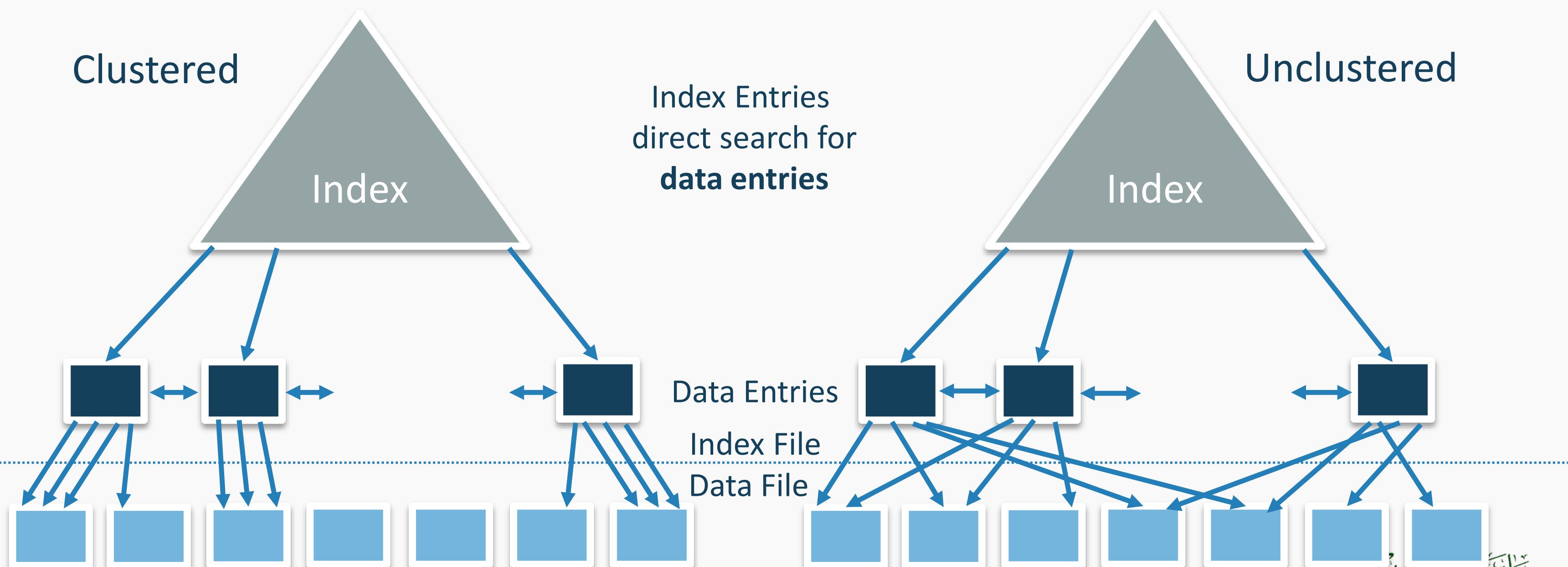
Clustered vs. Unclustered Index

- 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!
- Note: there is another definition of “clustering”
  - **Data Mining/AI**: grouping similar items in n-space

Clustered vs. Unclustered Index

Alternative 2: **Use references to data entries, data records in a Heap File**

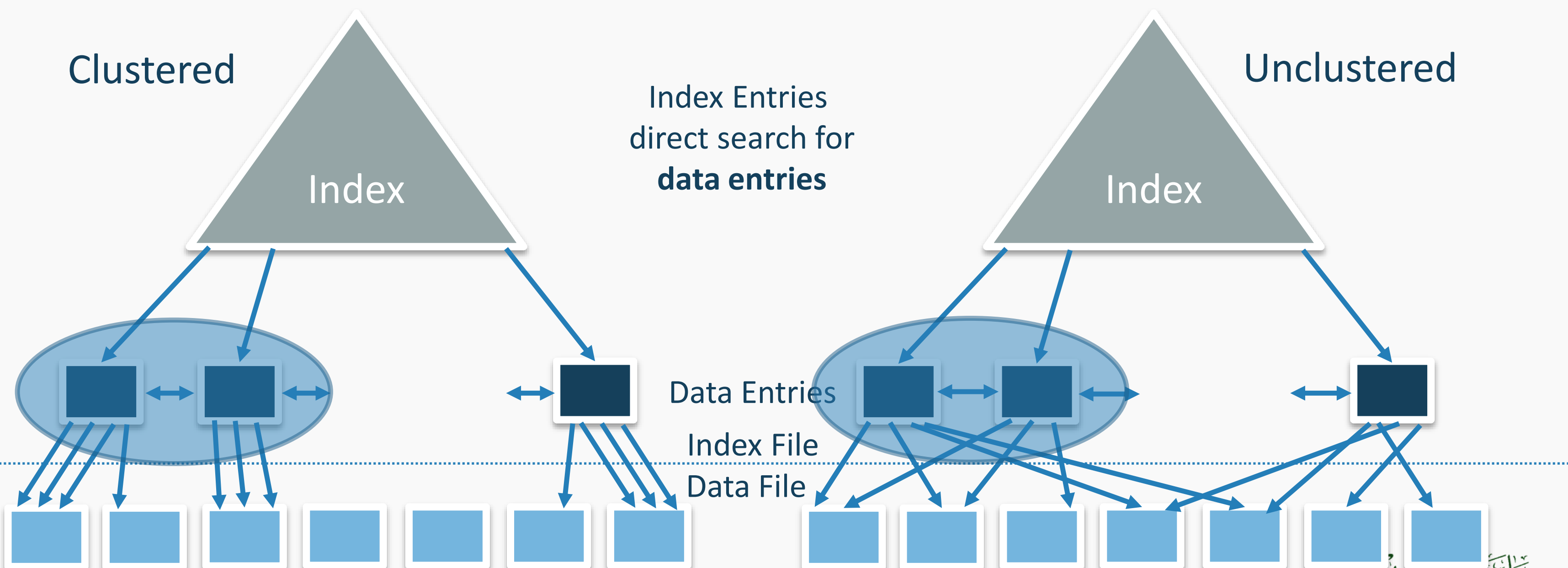
- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts
  - Thus, order of data records is "close to", but not identical to, the sort order



Clustered vs. Unclustered Index

Alternative 2: **Use references to data entries, data records in a Heap File**

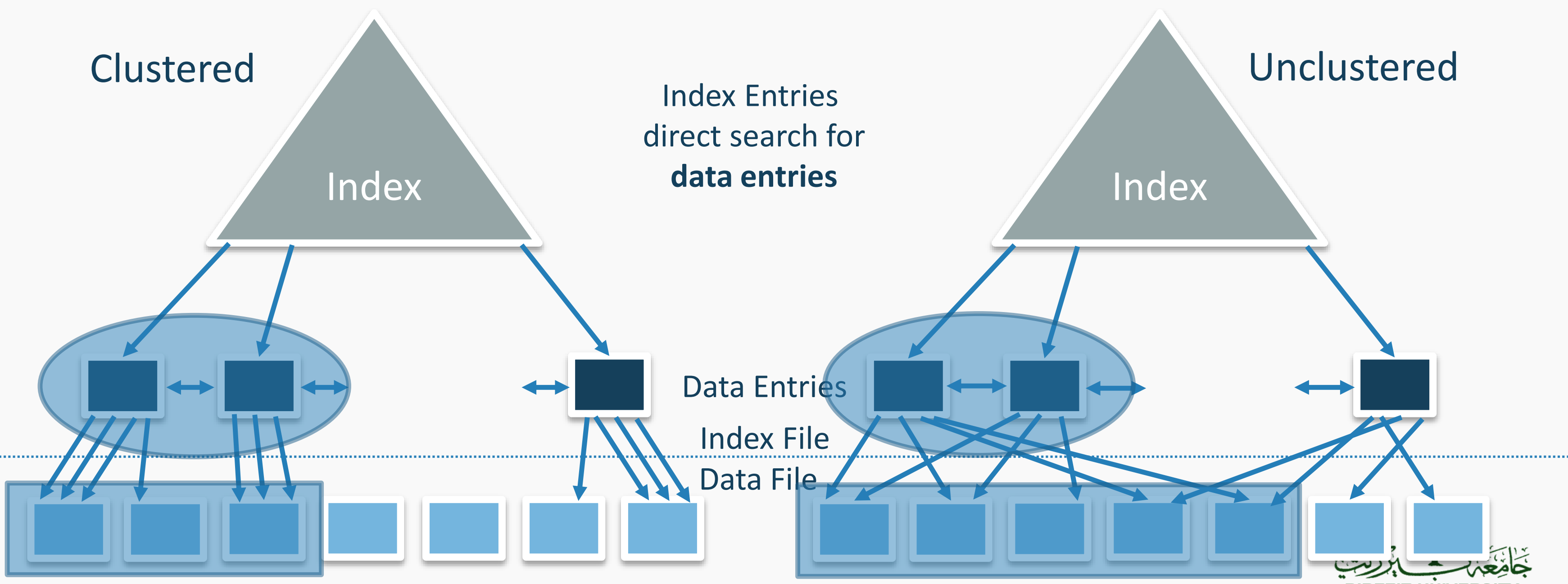
- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts
  - Thus, order of data records is “close to”, but not identical to, the sort order



Clustered vs. Unclustered Index

Alternative 2: **Use references to data entries, data records in a Heap File**

- To build a 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 records is “close to”, but not identical to, the sort order



Clustered vs. Unclustered Indexes

- Clustered Index Pros
  - Efficient for range searches
  - Potentially locality benefits?
    - Sequential disk access, prefetching, etc.
  - Support certain types of compression

Enhance compression algorithms. Graduation project or Master

- Clustered Cons
  - More expensive to maintain
    - Need to update index data structure
  - File usually only **packed to 2/3** to accommodate inserts
  - Need more storage space

Cost of Operations

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$ * D
Range Search	B*D	$((\log_2 B)$+pages)*D
Insert	2*D	$((\log_2 B)$+B)*D
Delete	(0.5*B+1)*D	$((\log_2 B)$+B)*D

Can we do better with indexes?

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations

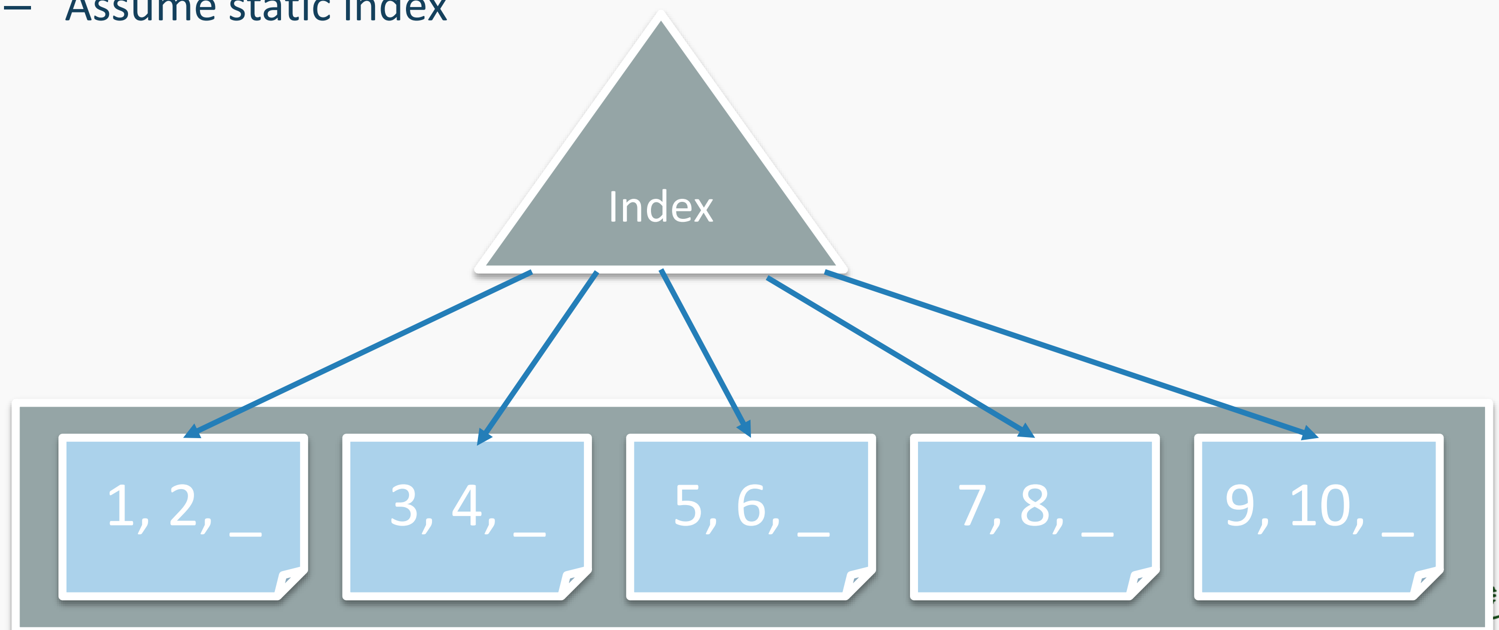
	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	
Equality Search	0.5*B*D	$(\log_2 B)$*D	
Range Search	B*D	$((\log_2 B)$+pages))*D	
Insert	2*D	$((\log_2 B) + B)$*D	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)$*D	

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Clustered vs. Unclustered Index

Assumptions:

- Store data by reference (Alternative 2)
- Clustered index with 2/3 full heap file pages
  - Clustered → Heap file is initially sorted
  - **Fan-out** (F): relatively large. Why?
    - Page of <key, pointer> pairs ~ O(R)
  - Assume static index



Cost of Operations

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	
Equality Search	0.5*B*D	$(\log_2 B)*D$	
Range Search	B*D	$((\log_2 B)+pages))*D$	
Insert	2*D	$((\log_2 B) + B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

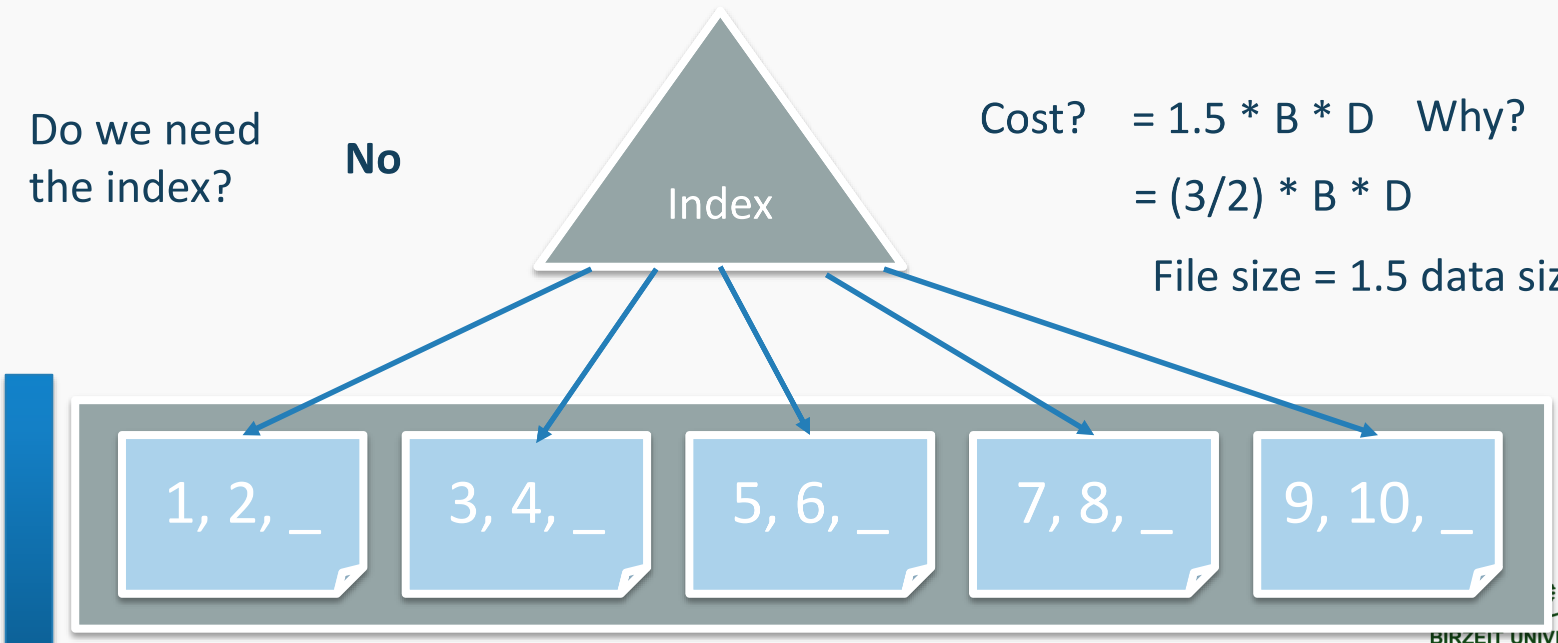
Scan all the Records

Assumptions:

- Store data by reference (Alternative 2)
- Clustered index with 2/3 full heap file pages
- Occupancy = 66.6%
  - Clustered → Heap file is initially sorted

Do we need the index?

No



Cost? = 1.5 * B * D Why?

= (3/2) * B * D

File size = 1.5 data size

Cost of Operations

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	1.5*B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D	
Range Search	B*D	$((\log_2 B)$+pages))*D	
Insert	2*D	$((\log_2 B) + B)$*D	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)$*D	

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	1.5*B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D	
Range Search	B*D	(($\log_2 B$)+pages))*D	
Insert	2*D	(($\log_2 B$) + B)*D	
Delete	(0.5*B+1)*D	(($\log_2 B$) + B)*D	

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

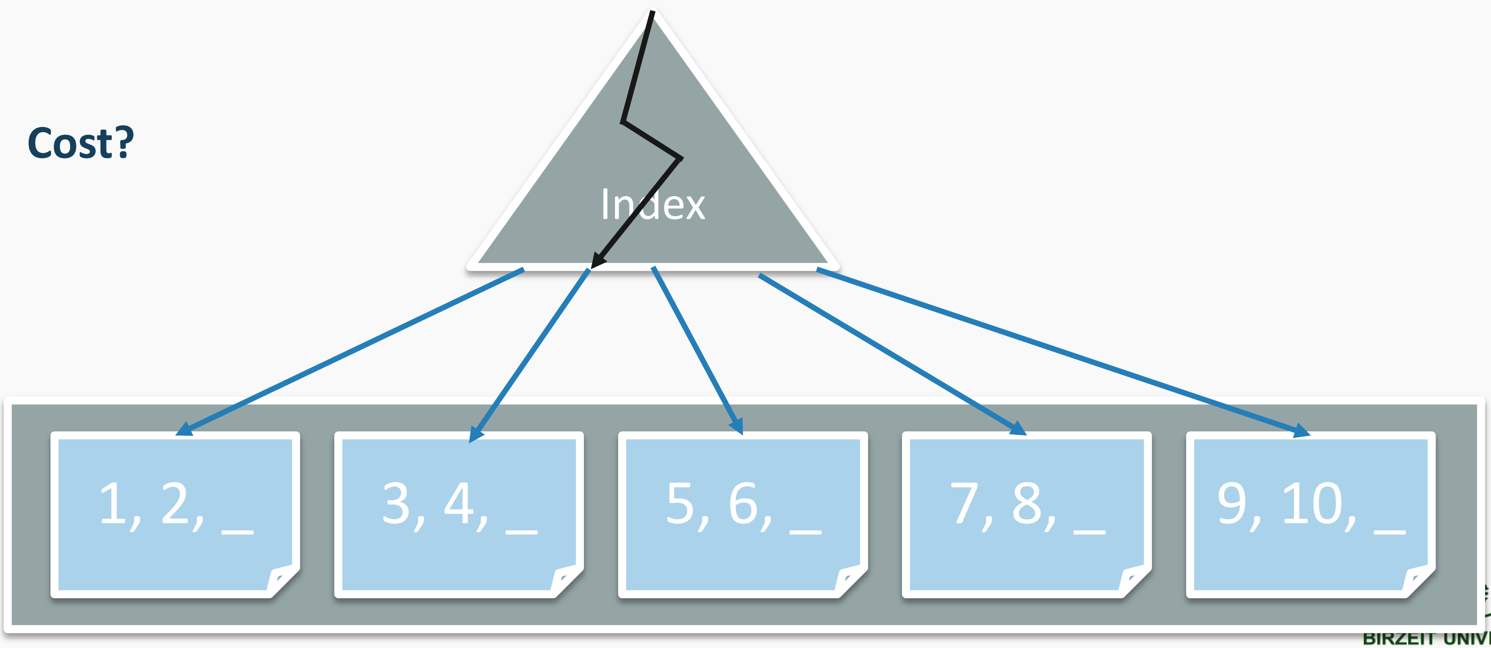
Find the record with key 3

Search the index: $= \log_F (1.5 * B) * D$

- Each page load narrows search by **factor of F**

- Lookup record in heap file by record-id = D

Cost?



Cost of Operations

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	1.5*B*D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$((\log_F 1.5*B))*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	
Insert	2*D	$((\log_2 B) + B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	1.5*B*D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$((\log_F 1.5*B))*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	
Insert	2*D	$((\log_2 B)+B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B)+B)*D$	

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

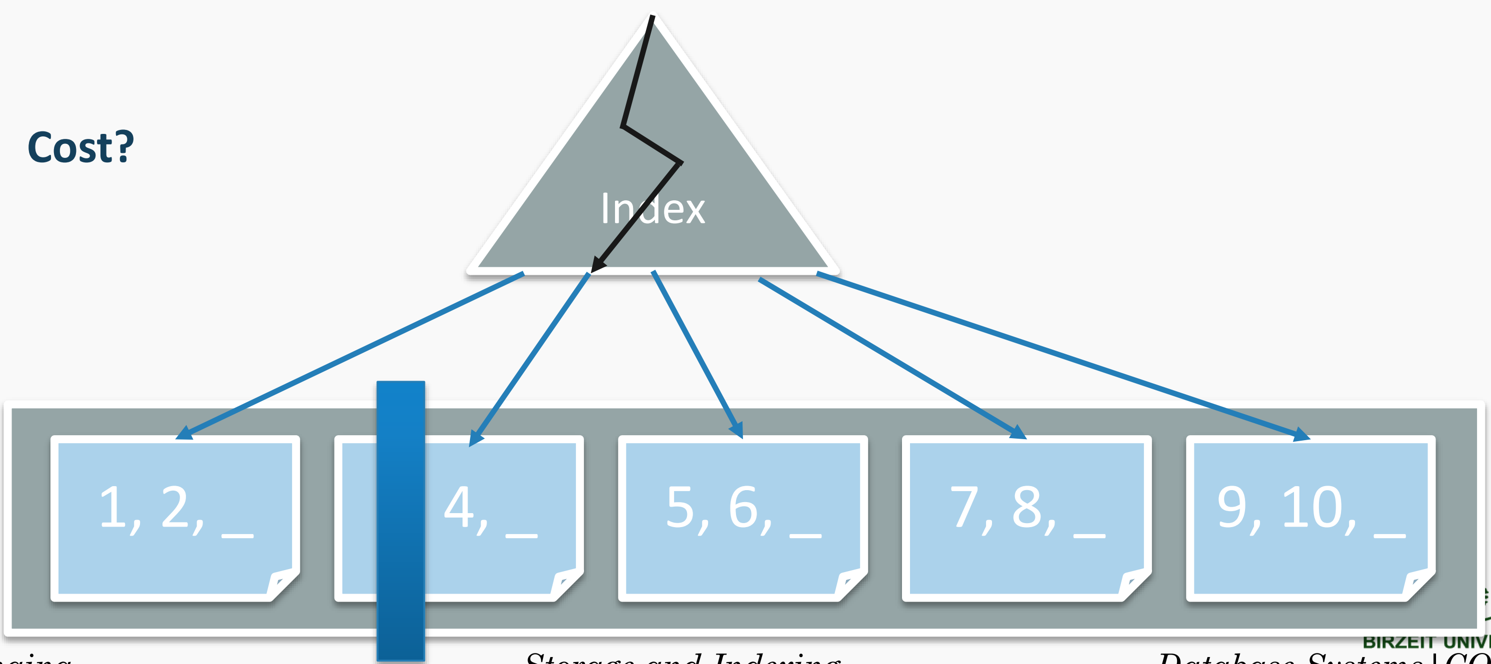
Find keys between 3 and 7

Search the index: $= \log_F (1.5 * B) * D$

- Each page load narrows search by **factor of F**
- Lookup record in heap file by record-id = D
- Scan the data pages until the end of range

= (#matching pages) * D

Cost?



Cost of Operations

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	1.5*B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D	$((\log_F 1.5$*B))*D
Range Search	B*D	$((\log_2 B)$+pages))*D	$((\log_F 1.5$*B)+pages)*D
Insert	2*D	$((\log_2 B)$+B)*D	
Delete	(0.5*B + 1)*D	$((\log_2 B)$+B)*D	

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	1.5*B*D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F 1.5*B)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	$((\log_F 1.5*B)+pages)*D$
Insert	2*D	$((\log_2 B)+B)*D$	
Delete	(0.5*B+1)*D	$((\log_2 B)+B)*D$	

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	1.5*B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D	$(\log_F 1.5$*B)*D
Range Search	B*D	$((\log_2 B)$+pages))*D	$((\log_F 1.5$*B)+pages)*D
Insert	2*D	$((\log_2 B)$+B)*D	$((\log_F 1.5$*B)+2)*D
Delete	(0.5*B+1)*D	$((\log_2 B)$+B)*D	

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	1.5*B*D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F 1.5*B)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	$((\log_F 1.5*B)+pages)*D$
Insert	2*D	$((\log_2 B)+B)*D$	$((\log_F 1.5*B)+2)*D$
Delete	(0.5*B+1) * D	$((\log_2 B)+B)*D$	$((\log_F 1.5*B)+2)*D$

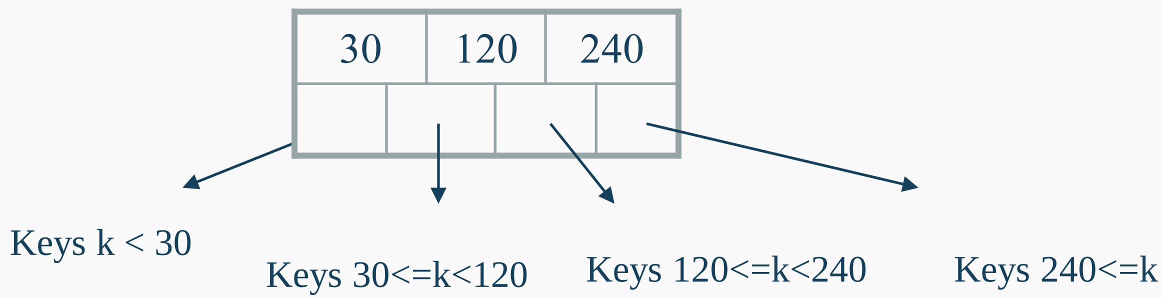
- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Tree-Based Indexing

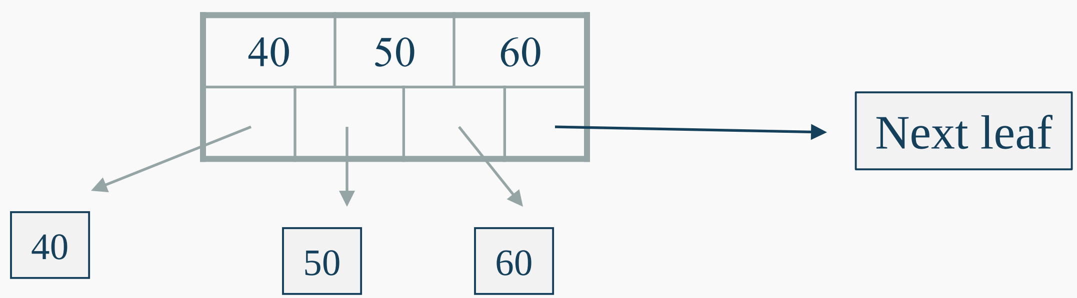
- Usually B+ tree is used.
- Each node points to one block
  - Make leaves into a linked list (range queries are easier)

B+ Trees Basics

- Parameter d = the degree
- Each node has >= d and <= 2d keys (except root)



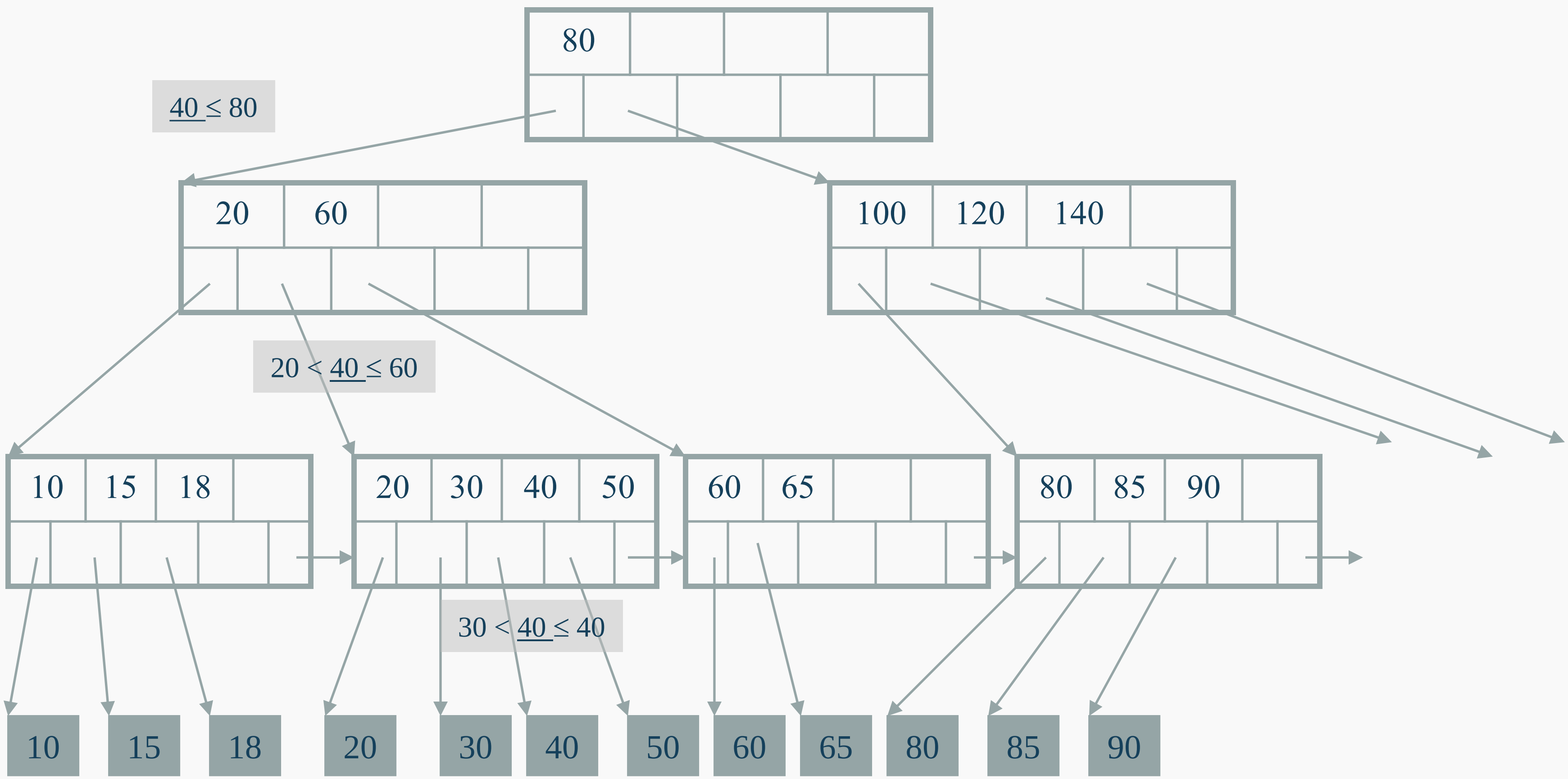
- Each leaf has >=d and <= 2d keys:



B+ Tree Example

d = 2

Find the key 40



Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
Select name
From people
Where age = 25
```

- Range queries:
  - As above
  - Then sequential traversal

```
Select name
From people
Where 20 <= age
  and  age <= 30
```

B+ Trees in Practice

The average number of children for a non-leaf node is called the **fan-out** of the tree.

- Typical order: d= 100.
- Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- B-Trees – dynamic, good for changing data, range queries

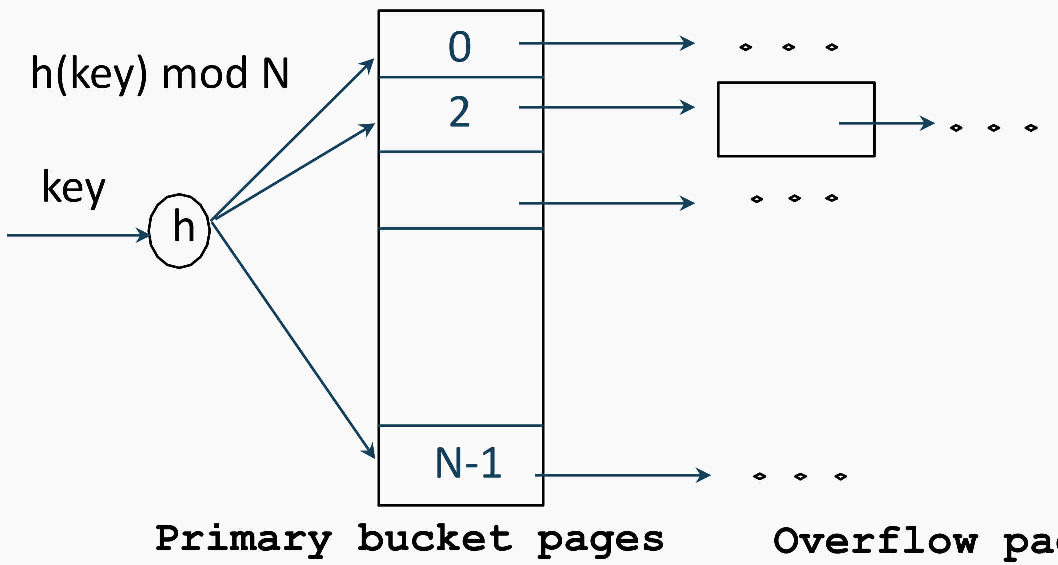
How many I/O needed to search for a record within 312 million records?

Hash-Based Indexes

- Good for equality selections.
- Index is a collection of *buckets.*
- Bucket = *primary* page plus zero or more *overflow* pages.
- Buckets contain data entries.
- *Hashing function* **h**: **h**(*r*) = bucket in which (data entry for) record *r* belongs.
- **h** looks at the *search key* fields of *r.*

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated;
- overflow pages if needed.
- h(*k*)= k mod N = bucket to which data entry with key *k* belongs. (N = # of buckets)
  - **h**(*k*) = (a * *k* + b) usually works well.
  - a and b are constants



Summary

- Many file organizations, with tradeoffs
  - Heap Files, Sorted Files, Clustered Files and Indexes
  - Benefits depend on the common operations
  - Compute expected costs
- Indexes: fast lookup of data entries by search key
  - Lookup: equivalence, range, region …
  - Search key: arbitrary columns
- Data Entries:
  - 3 alternatives: By Value, By Reference, By List of References

Summary

- Often multiple indexes per file of data records
  - Each with a different search key
- Indexes can be classified as clustered vs unclustered
  - Important consequences for utility/performance

Summary

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	$D\log_2 B$	$D(\log_2 B)$ +D. # pgs w. match recs	Search + BD	Search +BD
(3) Clustered	1.5BD	$D\log_F 1.5B$	$D(\log_F 1.5B)$ + D. # pgs w. match recs	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	$D(1 + \log_F 0.15B)$	$D(\log_F 0.15B$ + # match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D