Advanced Database Systems

Spring 2024

Lecture #13:

File Organisations

R&G: Chapter 8

Recap: File Organisations

Method of arranging a file of records on secondary storage

Heap Files

Store records in no particular order

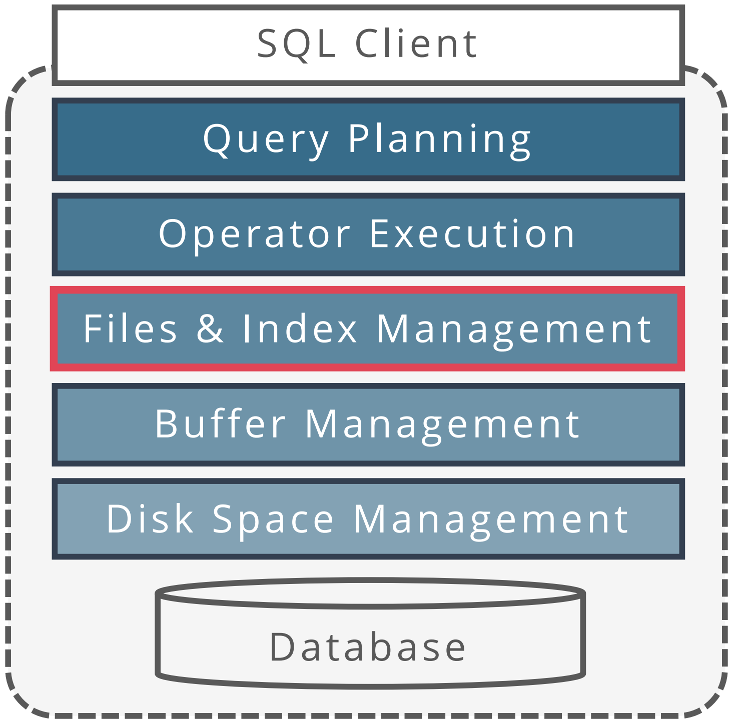
Sorted Files

Store records in sorted order, based on search key fields

Index Files

Store records to enable fast lookup and modifications

Tree-based & hash-based indexes



Comparing File Organisations

- What is the “best” file organisation?
  - Depends on access patterns...
  - What are common access patterns?
- How to compare file organisations anyway?
- Can we be quantitative about trade-offs?
  - If one is better ... by how much?

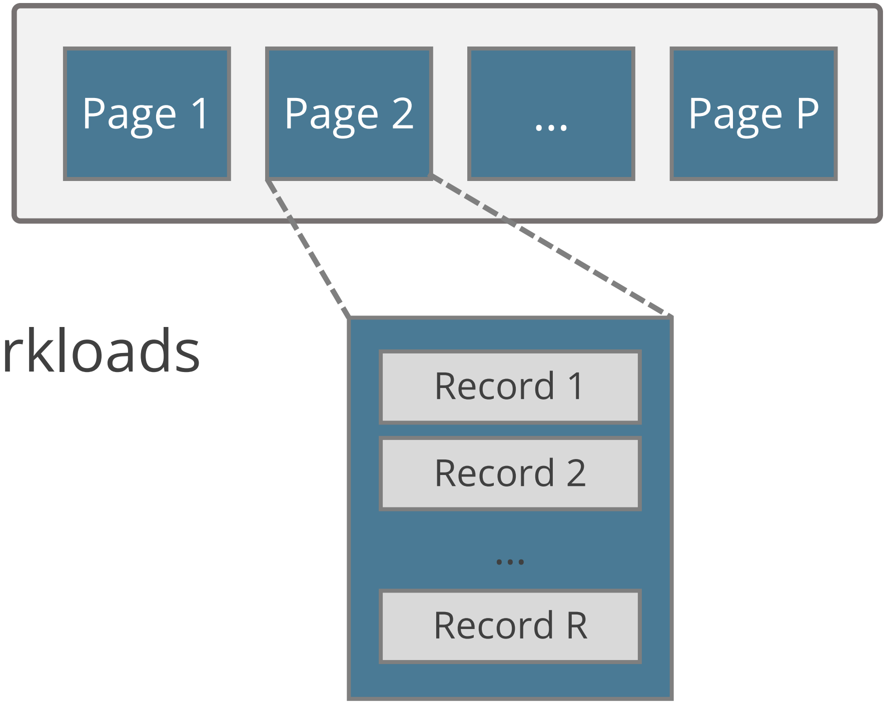
GOALS

- Big picture overheads for data access
  - We will (overly) simplify performance models to provide insight, not to get perfect performance
  - Still, a bit of discipline:
    - Clearly identify assumptions up front
    - Then estimate cost in a principled way
- Foundation for query optimization
  - Cannot choose the fastest scheme without an estimate of speed!

Cost Model for Analysis

Simplistic, but effective **I/O only cost model**

- **P** = Number of data pages in the file
- **R** = Number of records per page
- **D** = (Average) time to read or write disk page



Focus: Average case analysis for uniform random workloads

For now, we will ignore

- Sequential vs random I/O
- Prefetching pages
- Any in-memory costs (CPU cost is "free")

Good enough to show the overall trends

RECORD OPERATIONS

Scan all records in given file

```
SELECT * FROM R
```

Search with equality test

```
SELECT * FROM R WHERE C = 42
```

- On **key** attribute (e.g., studentID): assume **exactly one match**
- On **non-key** attribute (e.g., age): may return multiple matches

Search with range selection

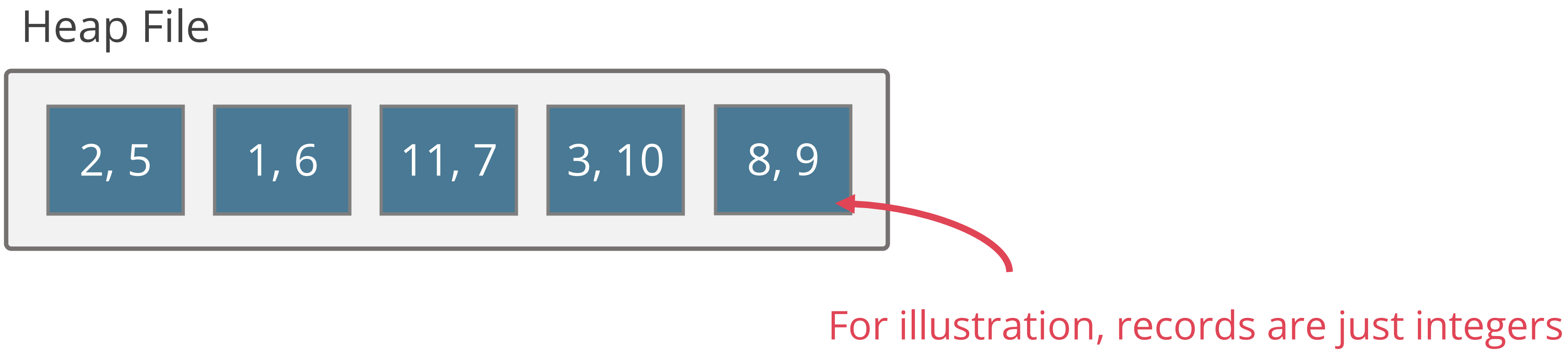
```
SELECT * FROM R WHERE A > 0 AND A < 100
```

Single record insert and delete

- For heap files, assume that insert always **appends to end of file**
- For sorted files, assume that files are compacted after deletions

Heap Files

A heap file maintains a collection of records in no particular order



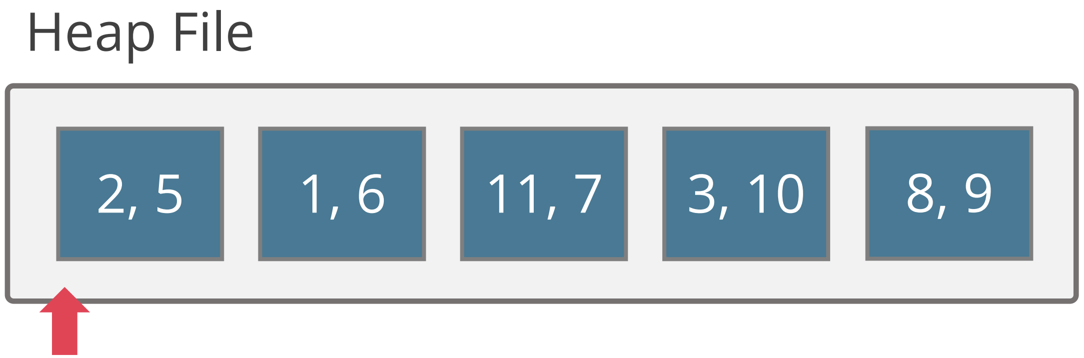
P: Number of data pages = 5

R: Number of records per page = 2

D: (Average) time to read/write disk page = 5 ms

Heap File: Scan All Records

Read each page of the file, for each page scan over all records



Scanning all records touches **P** pages

Reading each page takes **D** time

Estimated cost: P · D

Heap File: Search on Key

Search on key attribute ⇒ at most one match

Our assumption: searched record exists in the file (i.e., exactly one match)

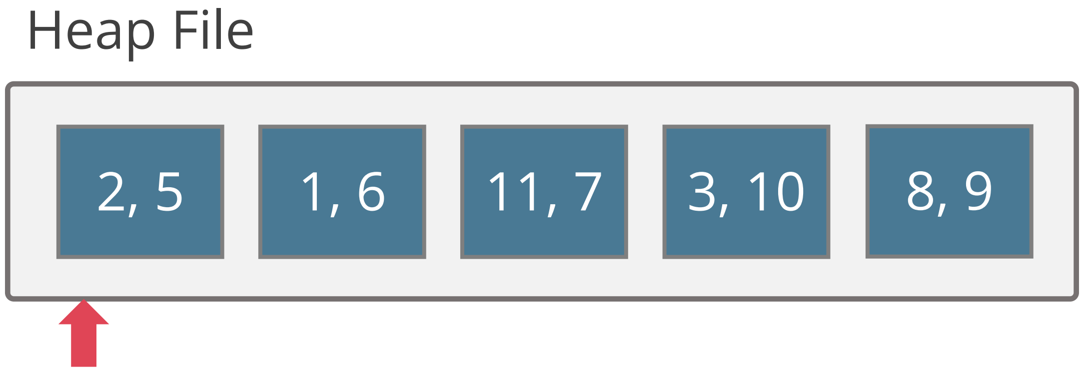
Pages touched on average?

Probability that key is on page **i** is **1/P**

If key is on page **i**, need to read **i** pages

Expected number of touched pages:

$$\sum_{i=1}^{P} i \, \frac{1}{P} = \frac{1}{P} \frac{P(P+1)}{2} = \frac{P+1}{2} \approx \frac{P}{2}$$



Cost: **P/2 · D**

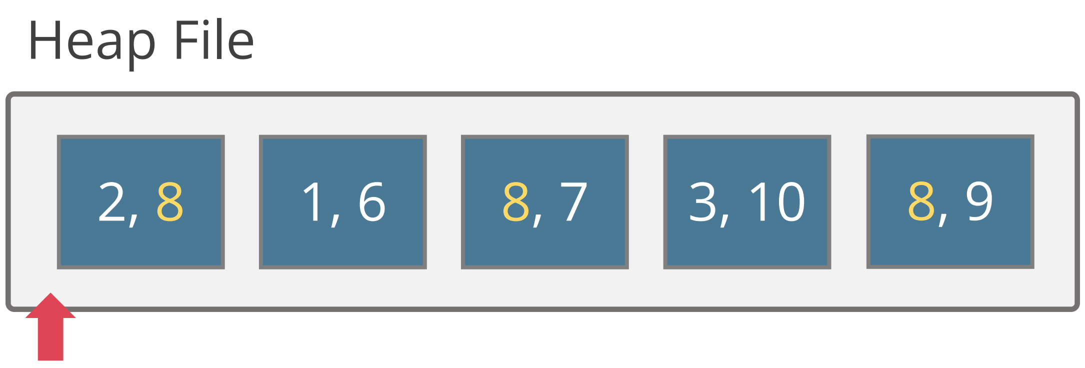
Heap File: Search on Non-Key

Search on non-key attribute ⇒ possibly multiple matches

- E.g.: Search for all records with value 8

Scan <u>all</u> pages in the file

- Records are stored in no particular order, thus we need to scan until the end of file to return all matching records



Cost: $P \cdot D$

Heap File: Range Search

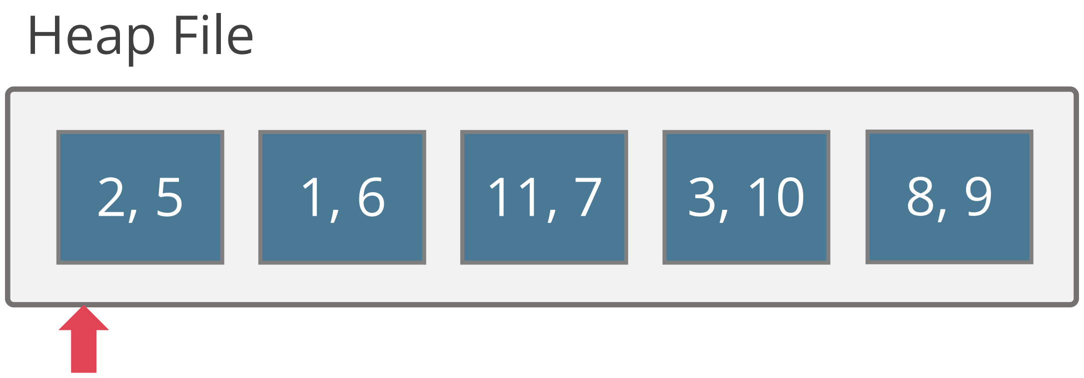
Range search

E.g.: Find records with values between 3 and 5

Always touch <u>all</u> pages

Same reasons as with searching on non-key attributes

Cost: $P \cdot D$



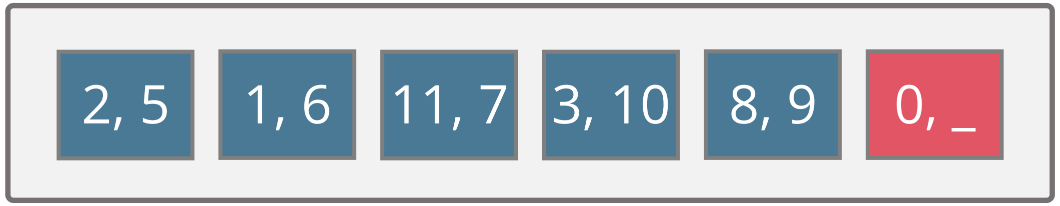
Heap File: Insert & Delete

Insert record

Read last page, append new record, write page back to disk

Cost = 2D

After Inserting 0



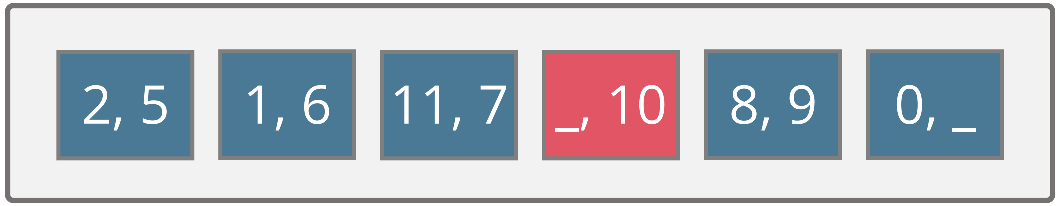
Delete record

Average case to find the record: P/2 reads

Delete record from page, write page back to disk

Cost = $(P/2 + 1) \cdot D$

After Deleting 3



Note: Records from last page could be used to eliminate gaps caused by deletions (ignored here)

Heap File Analysis

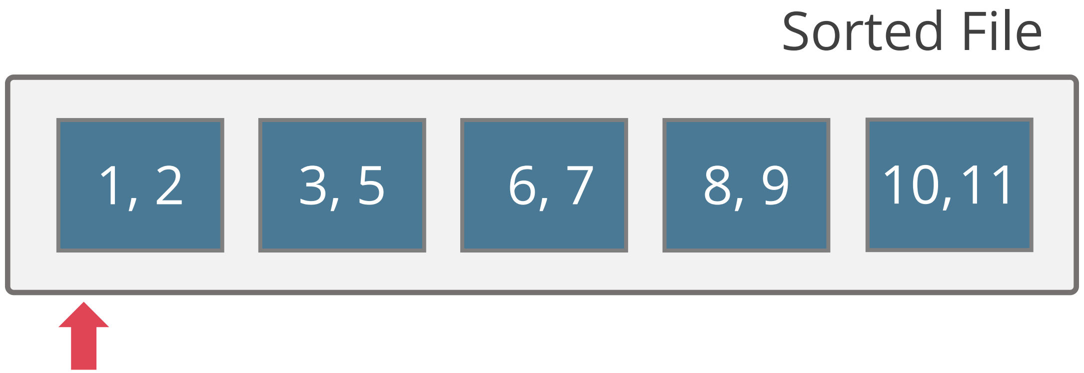
scan		iterate over all pages (linked or via directory)	$P \cdot D$
search	on key	scan the file until found	$0.5P \cdot D$
	on non-key	scan the file until end	$P \cdot D$
range query		same as scan	
delete		search, delete from page, and write page	$0.5P \cdot D + D$
insert		"just stick it at the end" (read last page, add, write)	$2D$

P = Number of data pages

D = (Average) time to read or write disk page

Sorted Files

Store records sorted by lookup attributes, no gaps



Scanning all records

Records in sorted order (can be big plus)

Cost: $P \cdot D$

Searching records

Use binary search when the **search condition matches the sort order**

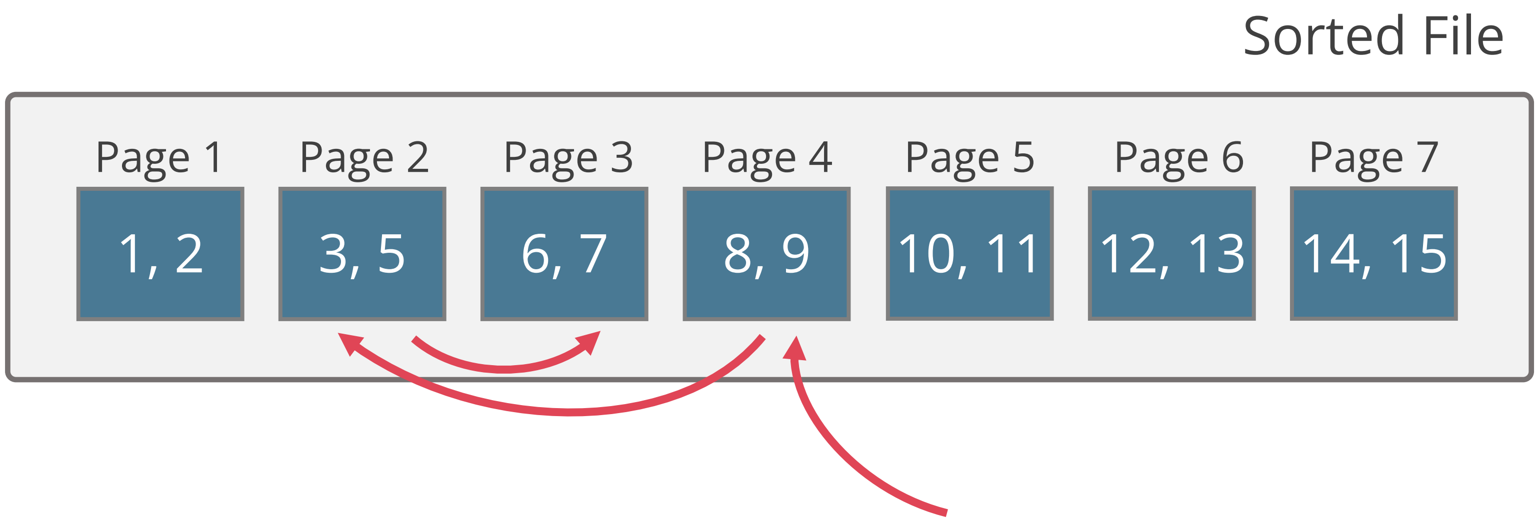
Inserts & deletes are **slow**

Shift all subsequent records

Sorted File: Search

Use binary search to locate matching record

- E.g.: Find record with value 6



Pages touched in binary search:

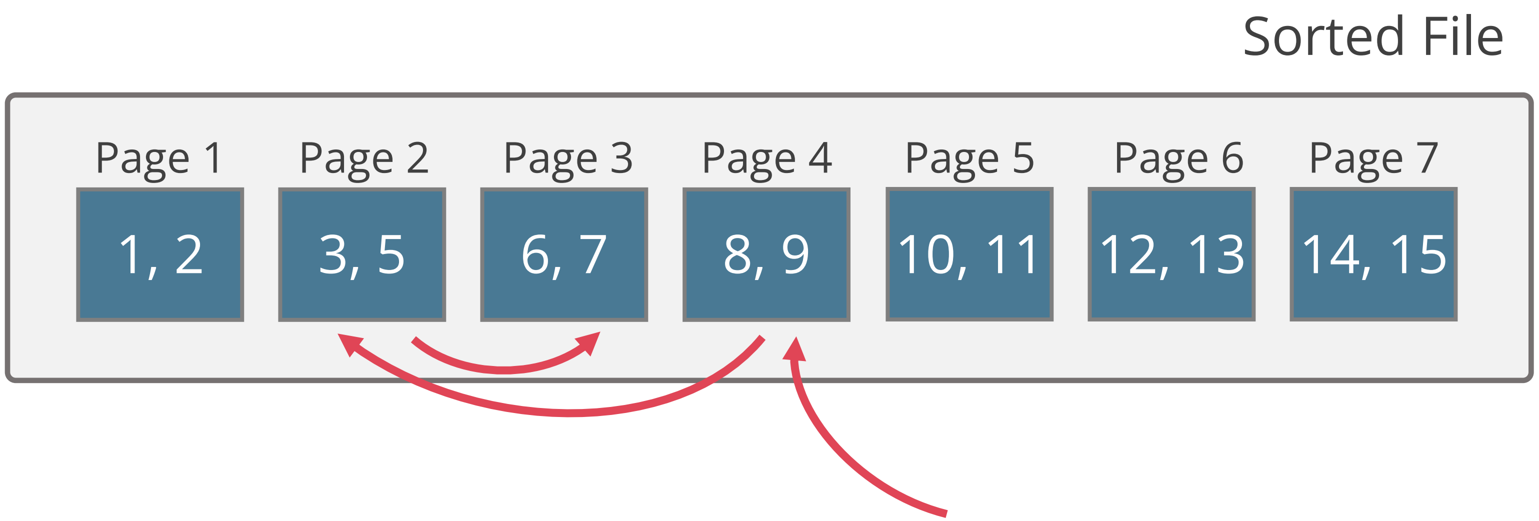
- Worst-case: **$\log_2 P$**
- Average-case: **$\log_2 P$**

Sorted File: Search

Binary search can be used only if the file is sorted on the search attribute

Search on key attribute

Cost: $\log_2 P \cdot D$



Search on non-key attribute & range search

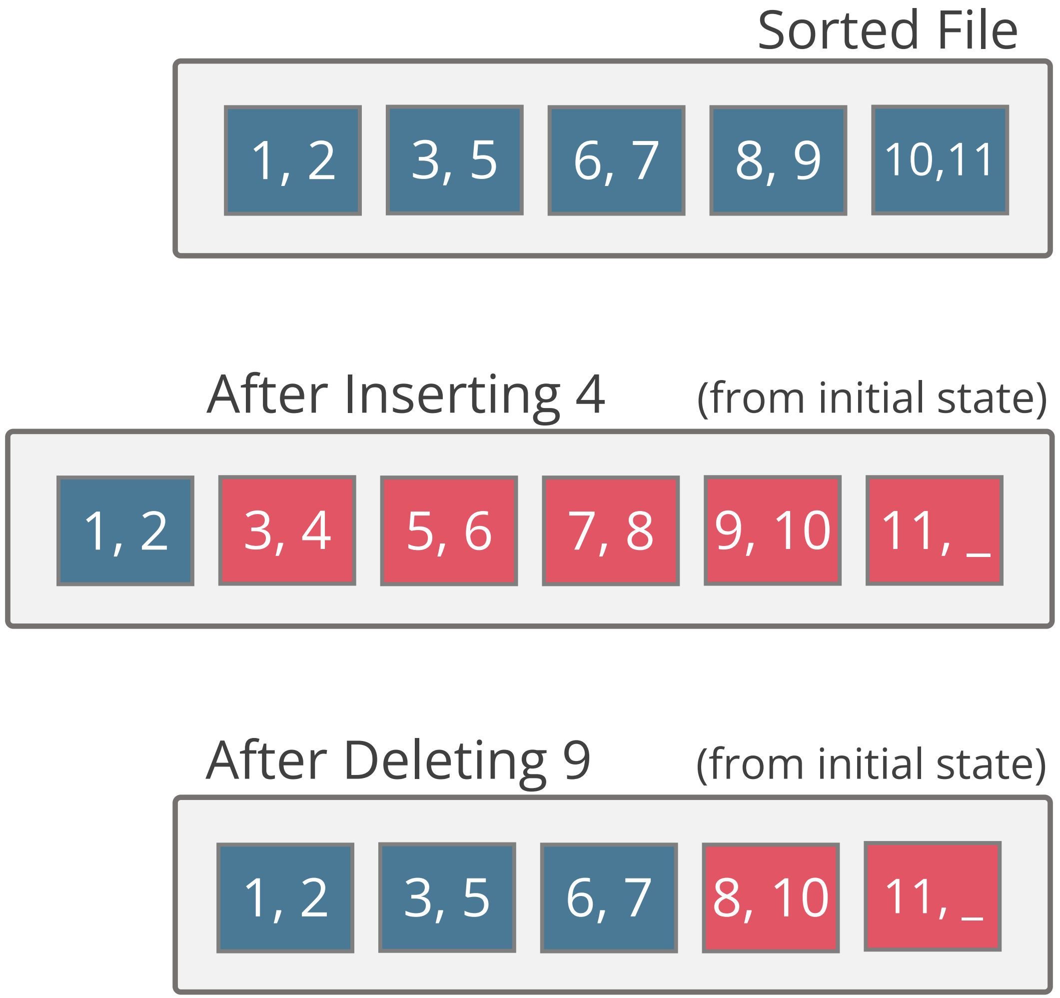
Search for start of range

Scan on right

Sorted File: Insert & Delete

Insert record

- Find location for record: $\log_2 P \cdot D$
- Insert and shift rest of file
  - On average shift **P / 2** pages
  - Read & write each shifted page
  - Cost: $2 (P / 2) \cdot D = P \cdot D$



Delete record

- Find location for record: $\log_2 P \cdot D$
- Delete and shift the rest by 1 record: $P \cdot D$

Sorted File Analysis

scan		iterate over all pages	$P \cdot D$
search	on key	binary search	$\log_2 P \cdot D$
	on non-key	binary search and search all matching records	$D \cdot (\log_2 P + \textit{\# pgs with match recs})$
range query		similar as search on non-key	
delete		search, delete, shift	search + $P \cdot D$
insert		search, insert, shift	search + $P \cdot D$

P = Number of data pages

D = (Average) time to read or write disk page

Heap File vs. Sorted File

		Heap File	Sorted File
scan		$P \cdot D$ 	$P \cdot D$ 
search	on key	$0.5P \cdot D$ 	$\log_2 P \cdot D$ 
	on non-key	$P \cdot D$ 	$(\log_2 P + \textit{\# pgs with match recs}) \cdot D$ 
range query		$P \cdot D$ 	$(\log_2 P + \textit{\# pgs with match recs}) \cdot D$ 
delete		$0.5P \cdot D + D$ 	$(\log_2 P + P) \cdot D$ 
insert		$2D$ 	$(\log_2 P + P) \cdot D$ 

P = Number of data pages

D = (Average) time to read or write disk page

 = very fast

 = fast

 = slow

Can we do better? Yes, indexes!

Introduction to Indexes

Index = data structure that enables fast lookup by value

You may have seen in-memory data structures in Algorithms & DS course

- Search trees (Binary, AVL, Red-Black, …)
- Hash tables (Chained, Open Addressing, …)

Needed: persistent disk-based data structures

- "Paginated": made up of disk pages
- Cost of **access** & **maintenance** measured in I/Os

Our focus: disk-based indexes

- Tree-based & hash-based

INDEX

Index = data structure that organizes data records to efficiently retrieve all records matching a given **search condition**

Search key = attributes used to look up records in a relation

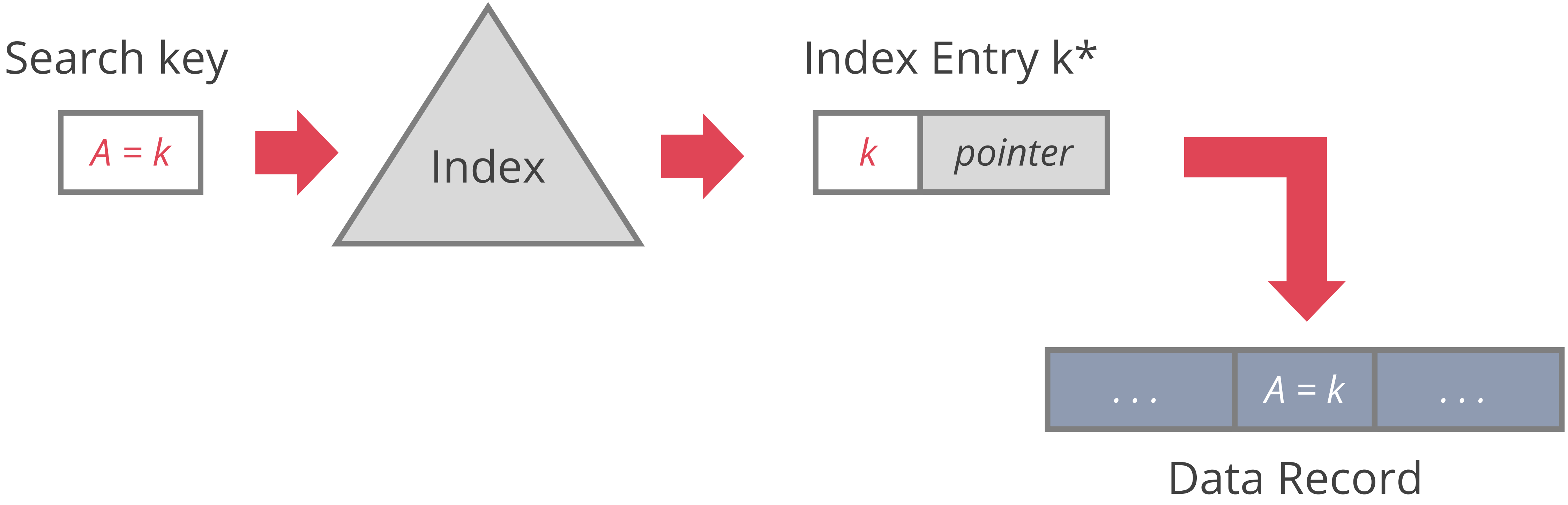
Can be any subset of the attributes of a relation. Do not need to be unique

Not the same as **key** = minimal set of attributes that uniquely identify a record

```
CREATE INDEX idx1 ON Student USING btree(sid)
CREATE INDEX idx2 ON Student USING hash(sid)
CREATE INDEX idx3 ON Student USING btree(age,name)
```

Index Usage

An index contains a collection of **index entries** and supports efficient retrieval of all index entries **k*** with a given key value **k**



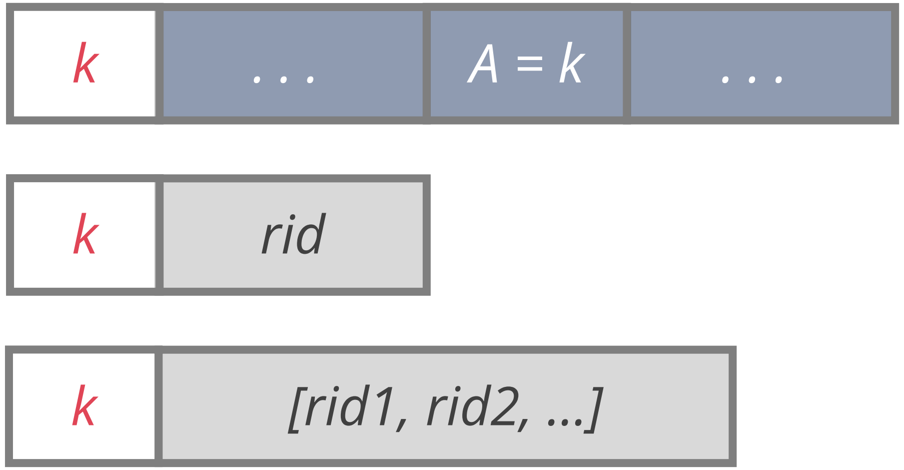
INDEX ENTRIES

We can design the index entries (**k***) in various ways

Variant A By Value

Variant B By Reference

Variant C By List of References



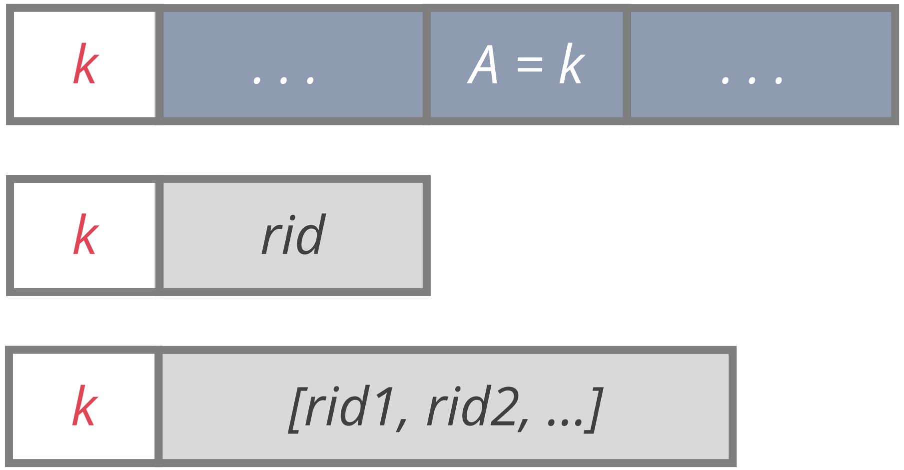
A: Record contents are stored in the index file

To avoid redundant storage of records at most one index on a table can use A

B and C use record IDs (rids) to point into the actual data file (heap file)

Index Entries

Variant choice is orthogonal to the type of index (B+ trees, hash)



Variant A	k | . . . | A = k | . . .	By Value
Variant B	k | rid	By Reference
Variant C	k | [rid1, rid2, ...]	By List of References

B and **C** have index entries typically much smaller than data records

C leads to less index entries if multiple records match a search key *k*, but index entries are of variable length

Indexing by Reference

Both **Variant B** and **Variant C** index data by reference

By-reference is required to support multiple indexes per table

- Otherwise we would be replicating entire tuples
- Replicating data leads to complexity when doing updates,
  so it's something we want to avoid

Index Classification

Tree-based vs. Hash-based

Do not support same operations: range search only in tree indexes

Clustered vs. unclustered

Clustered = order of data records is same as, or 'close to', order of index entries

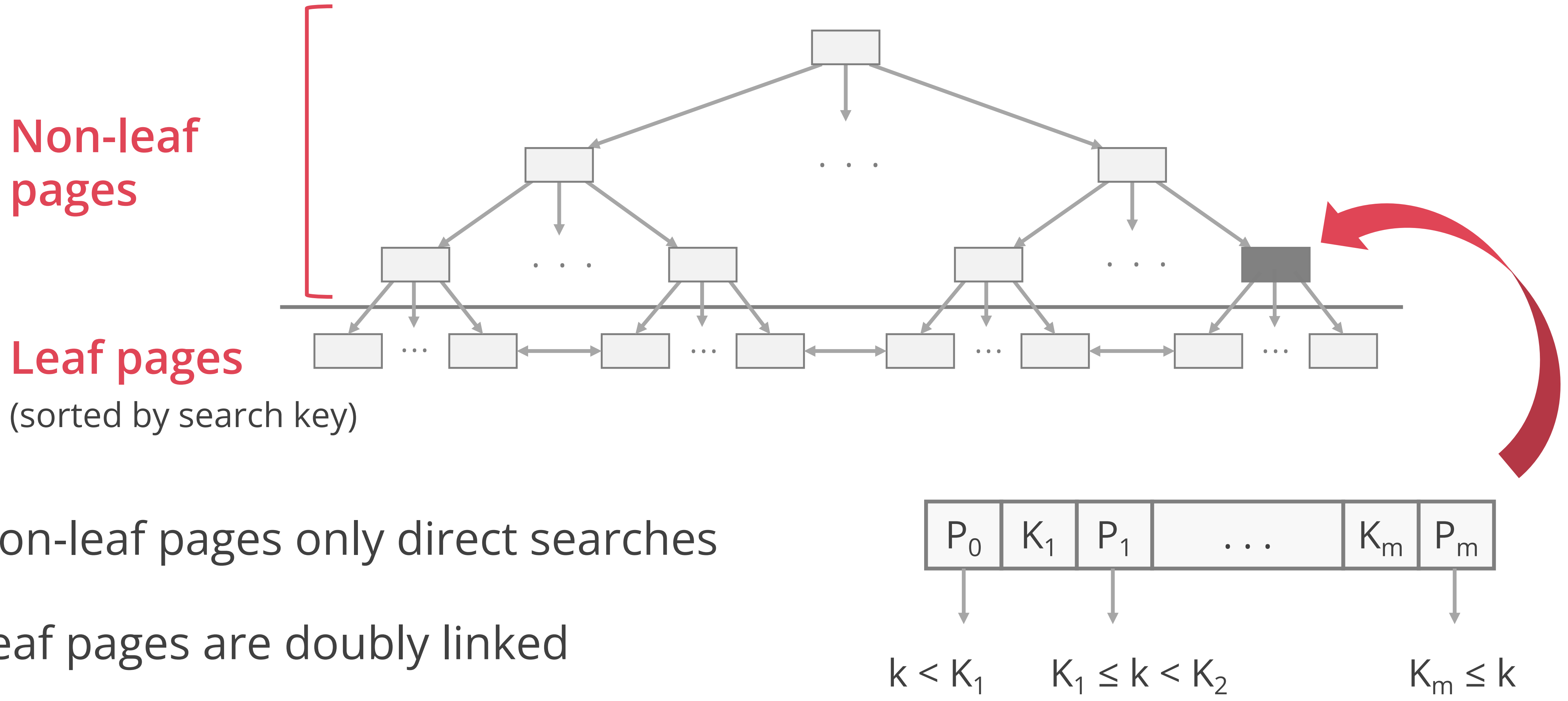
Primary vs. secondary

Primary index = search key contains primary key

Secondary index = search key may match multiple records

Unique index = search key contains a candidate key

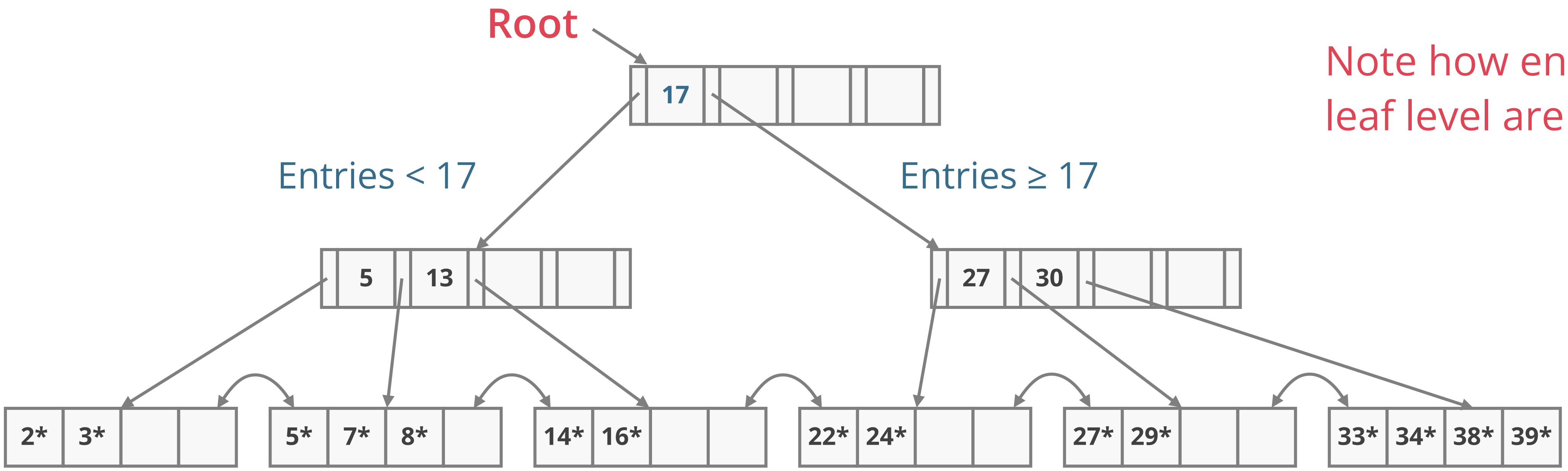
B+ Tree Indexes



Non-leaf pages only direct searches

Leaf pages are doubly linked

Example B+ Tree



Note how entries in leaf level are sorted

Find 29*? Find 28*? Find all entries > 15* and < 30*

Insert/delete: Find data entry in leaf, then change it

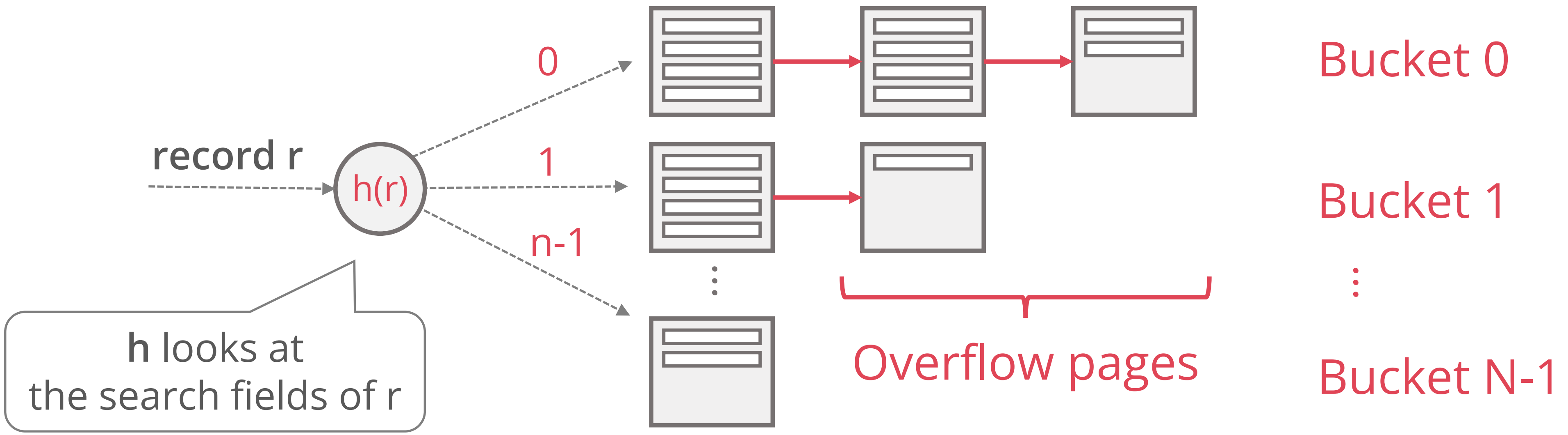
Need to adjust parent sometimes. Change sometimes bubbles up the tree

Hash-Based Index

Index is a collection of **buckets**

- Bucket = **primary page** plus zero or more **overflow pages**
- Buckets contain index entries

Hashing function h: h(r) = bucket in which record r belongs



Clustered vs. Unclustered Index

By-reference indexes (Variants B and C) can be **clustered** or **unclustered**

- Really this is a property of the heap file associated with the index!

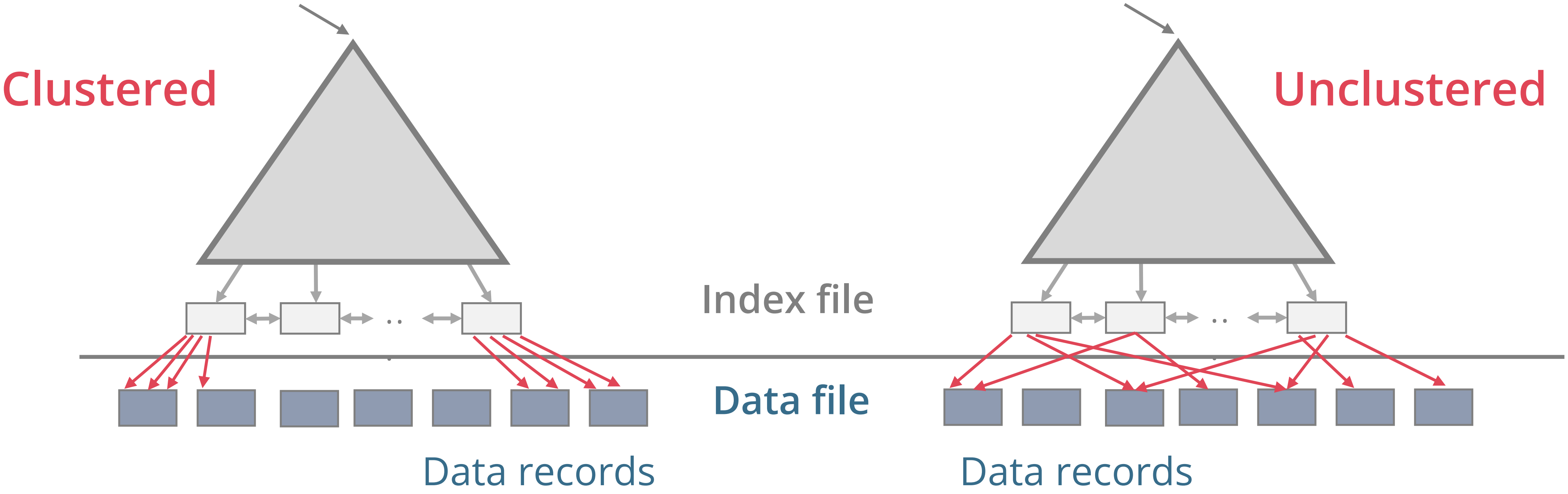
Clustered index

- Heap file records are kept mostly ordered according to **search key** in index
  - Heap file order need not be perfect: this is just a performance hint
- Cost of retrieving records can **vary greatly** based on whether index is clustered or not!

A heap file can be clustered on at **most one** search key

Variant A implies clustered index

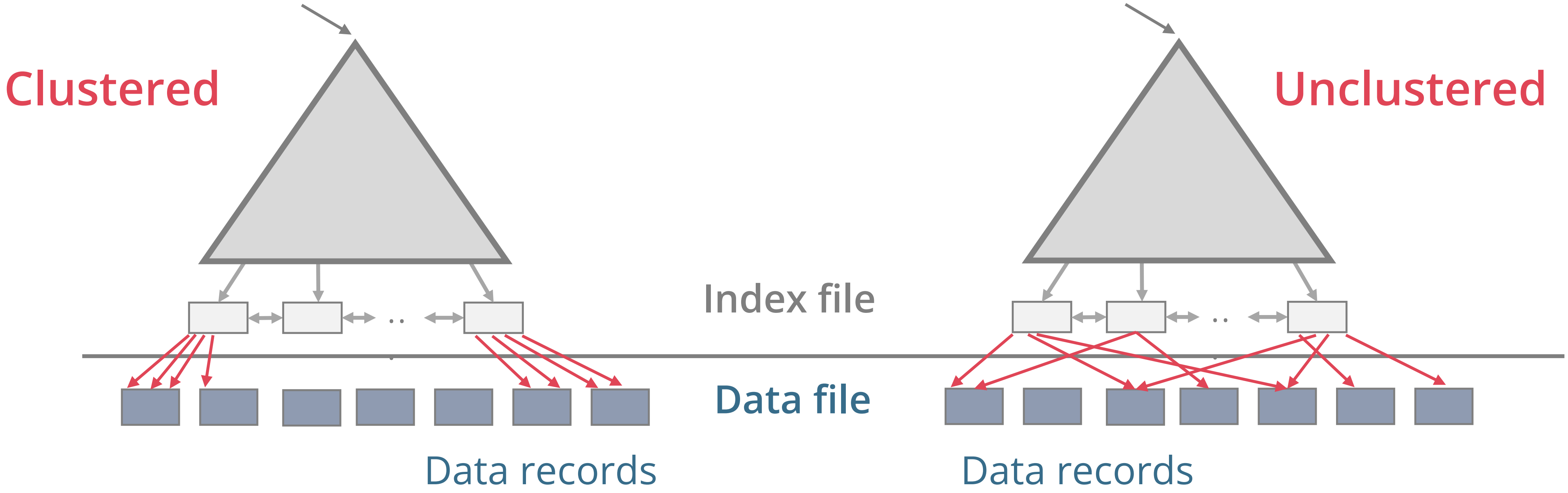
CLUSTERED VS. UNCLUSTERED INDEX



To build a clustered index, first sort the heap file

- Leave some free space on each page for future inserts
- Index entries direct search for data entries

CLUSTERED VS. UNCLUSTERED INDEX



Cost of retrieving records can vary greatly

- Clustered: I/O cost = # pages in data file with matching records
- Unclustered: I/O cost ≈ # of matching **leaf index entries** (i.e., matching records)

Clustered vs. Unclustered Index

Clustered pros

- Efficient for range searches
- Potential locality benefits
  - Sequential disk access, prefetching, etc
- Support certain types of compression (more on that later)

Clustered cons

- More expensive to maintain
  - Need to periodically update heap file order
  - Solution: on the fly or “lazily” via reorganisations
- Heap file usually only **packed to 2/3** to accommodate inserts
  - Overflow pages may be needed for inserts

SUMMARY

Heap Files: Suitable when typical access is a full scan of all records

Sorted Files: Best for retrieval in order or when a range of records is needed

Index Files: Fast lookup and efficient modifications

- Tree-based vs. hash-based
  - Hash-based index only good for equality search
  - Tree index is always a good choice
- Clustered vs. unclustered index
  - At most one clustered index per table
  - Multiple unclustered indexes are possible