



# Advanced Database Systems

Spring 2026

Lecture #08:

## File Organisations

R&G: Chapter 8

If you require this document in an alternative format, such as large print or a coloured background, please contact [milos.nikolic@ed.ac.uk](mailto:milos.nikolic@ed.ac.uk)

1

## RECAP: FILE ORGANISATIONS

2

Method of arranging a file of records on secondary storage

### Heap Files

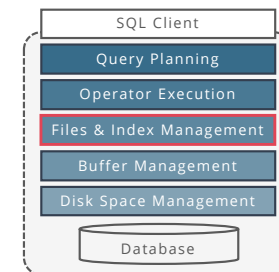
Store records in no particular order

### 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



2

## COMPARING FILE ORGANISATIONS

3

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?

3

## GOALS

4

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!

4

## COST MODEL FOR ANALYSIS

5

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

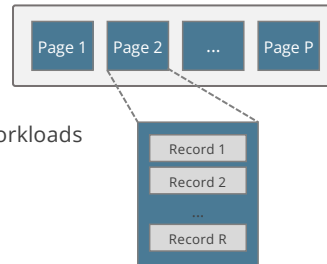
For now, we will ignore

Sequential vs random I/O

Prefetching pages

Any in-memory costs (CPU cost is “free”)

Good enough to show the overall trends



## RECORD OPERATIONS

6

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

6

## HEAP FILES

7

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

Heap File



For illustration, records are just integers

**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

8

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

Heap File



Scanning all records touches **P** pages

Reading each page takes **D** time

**Estimated cost:  $P \cdot D$**

8

5

7

## HEAP FILE: SEARCH ON KEY

9

Search on key attribute  $\Rightarrow$  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:  $P/2 \cdot D$

Heap File



## HEAP FILE: SEARCH ON NON-KEY

10

Search on non-key attribute  $\Rightarrow$  possibly multiple matches

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

Scan all 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



10

## HEAP FILE: RANGE SEARCH

11

Range search

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

Always touch all pages

Same reasons as with searching on non-key attributes

Cost:  $P \cdot D$

Heap File



## HEAP FILE: INSERT & DELETE

12

Insert record

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

Cost =  $2D$

After Inserting 0



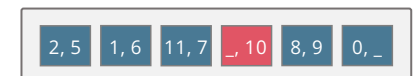
Delete record

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)

12

9

11

## HEAP FILE ANALYSIS

13

<b>scan</b>	iterate over all pages (linked or via directory)	$P \cdot D$
<b>search</b>	on key scan the file until found	$0.5P \cdot D$
	on non-key scan the file until end	$P \cdot D$
<b>range query</b>	same as scan	
<b>delete</b>	search, delete from page, and write page	$0.5P \cdot D + D$
<b>insert</b>	"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

14

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

13

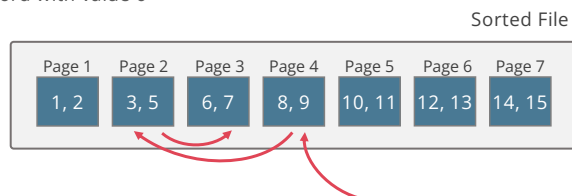
14

## SORTED FILE: SEARCH

15

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:  $\log_2 P$

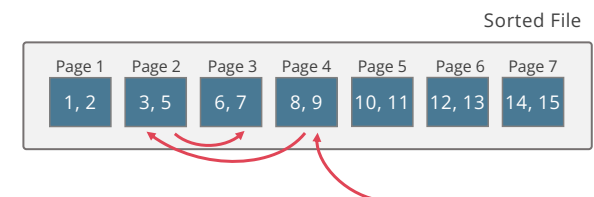
## SORTED FILE: SEARCH

18

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

15

18

## SORTED FILE: INSERT & DELETE

19

### Insert record

Find location for record:  $\log_2 P \cdot D$

Insert and shift rest of file

On average shift  $P/2$  pages

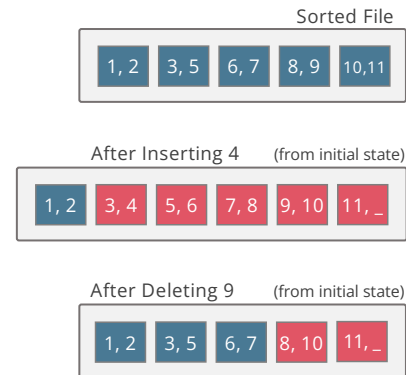
Read & write each shifted page

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

20

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

19

20

## HEAP FILE VS. SORTED FILE




21

	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	$P \cdot D$	$(\log_2 P + \# \text{ pgs with match recs}) \cdot D$	
range query	$P \cdot D$		$(\log_2 P + \# \text{ pgs with match recs}) \cdot D$	
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

Can we do better? Yes, indexes!

 = very fast  
 = fast  
 = slow

## INTRODUCTION TO INDEXES

22

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

21

22

## INDEX

23

**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

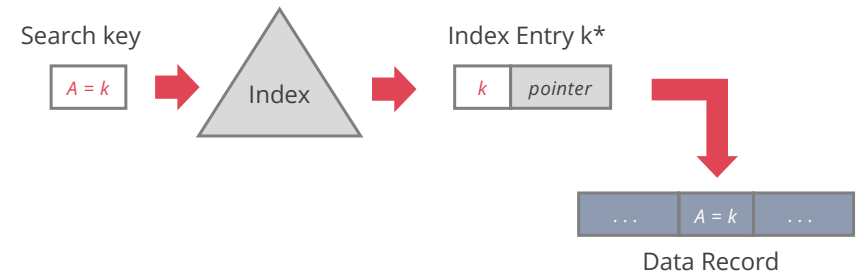
**Not** the same as **key** = minimal set of attributes that uniquely identify a record

```
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

24

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

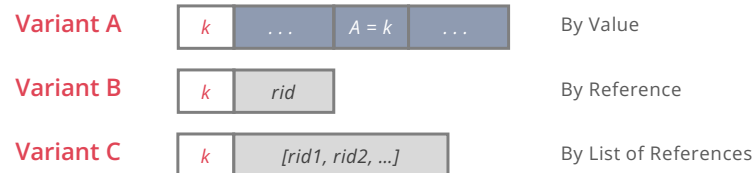


24

## INDEX ENTRIES

25

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



**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

26

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



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

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

23

25

26

## INDEXING BY REFERENCE

27

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

29

### 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

27

29

## B+ TREE INDEXES

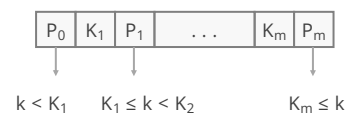
31

Non-leaf  
pages

Leaf pages  
(sorted by search key)

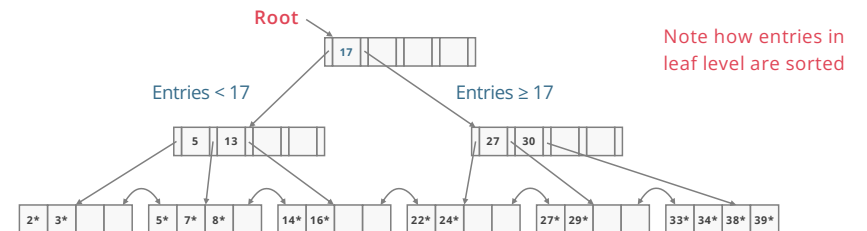
Non-leaf pages only direct searches

Leaf pages are doubly linked



## EXAMPLE B+ TREE

32



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

31

32

## HASH-BASED INDEX

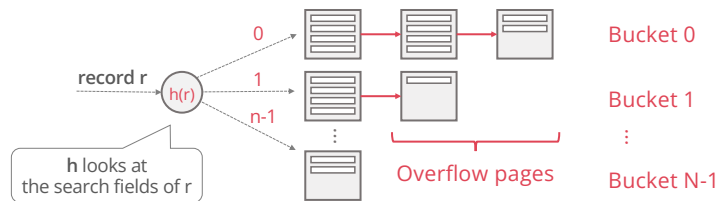
33

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



33

## CLUSTERED VS. UNCLUSTERED INDEX

36

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!

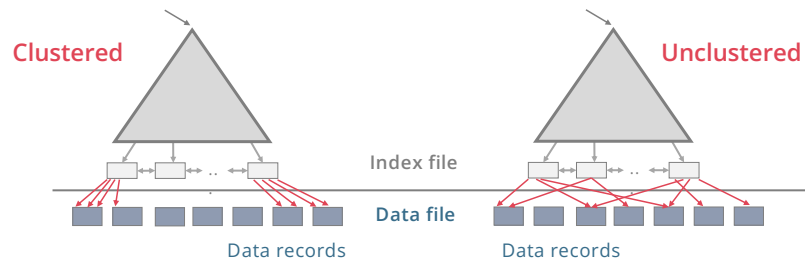
A heap file can be clustered on at **most one** search key

Variant **A** implies clustered index

36

## CLUSTERED VS. UNCLUSTERED INDEX

37



To build a clustered index, first sort the heap file

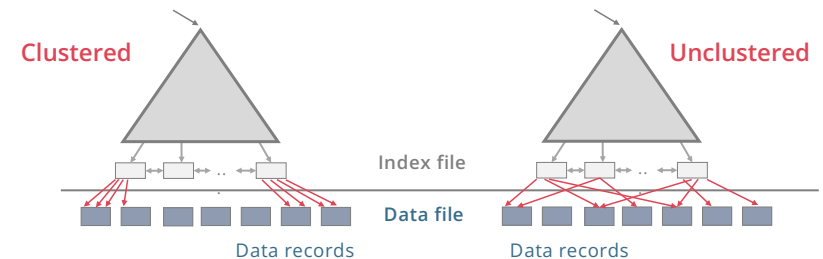
Leave some free space on each page for future inserts

Index entries direct search for data entries

37

## CLUSTERED VS. UNCLUSTERED INDEX

39



Cost of retrieving records can vary greatly

Clustered: I/O cost = # pages in data file with matching records

Unclustered: I/O cost  $\approx$  # of matching **leaf index entries** (i.e., matching records)

39



## CLUSTERED VS. UNCLUSTERED INDEX

40

### Clustered pros

- Efficient for range searches

- Potential locality benefits

  - Sequential disk access, prefetching, etc

- Support certain types of compression

  - Sorted data → high likelihood of repetitive values or patterns that compression algos can exploit

### 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

40

## SUMMARY

49

**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

49