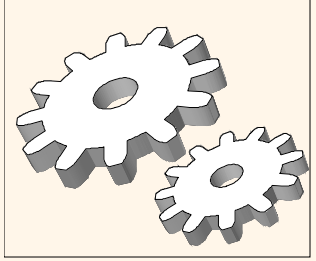
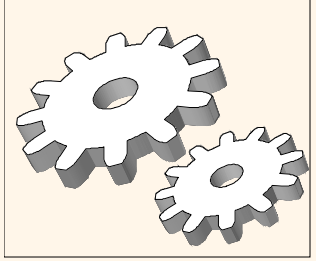


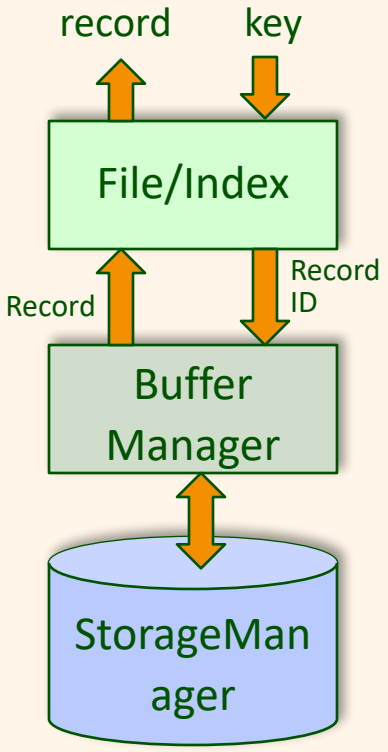
Overview of Storage and Indexing

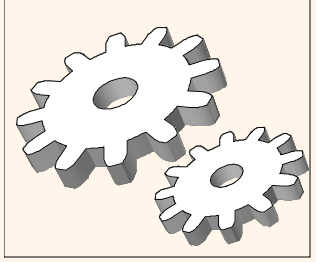
Chapter 8



Data on External Storage

- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order
- Tapes: Can only read pages in sequence
  - Cheaper than disks; used for archival 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
- Buffer Manager: stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

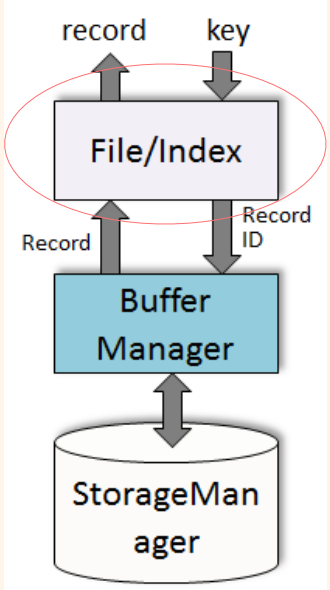




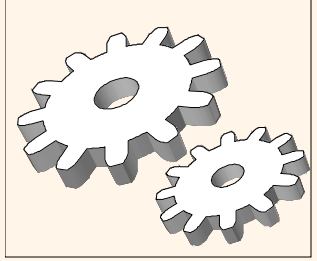
Alternative File Organizations

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

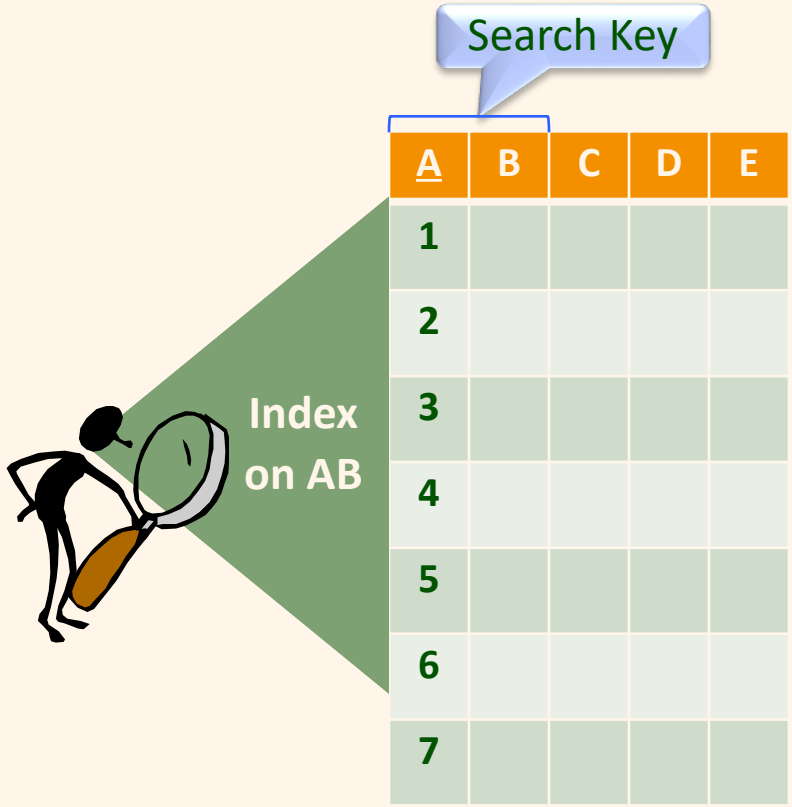
- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes: Data structures to organize records via **trees** or **hashing**.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - **Updates are much faster** than in sorted files.



Indexes – Search Key

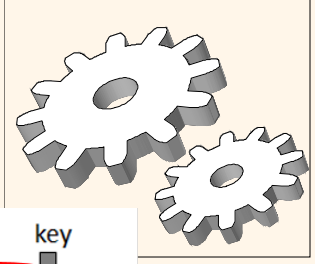


An *index* on a file speeds up selections on the *search key fields* for the index.



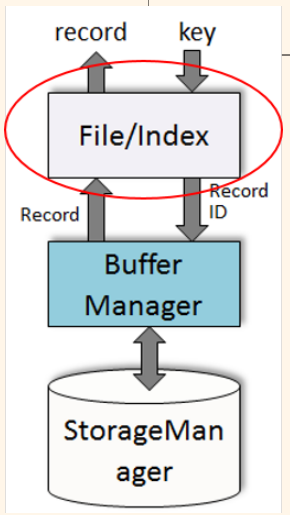
- Any subset of the fields of a relation can be the search key for an index on the relation.
- *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).

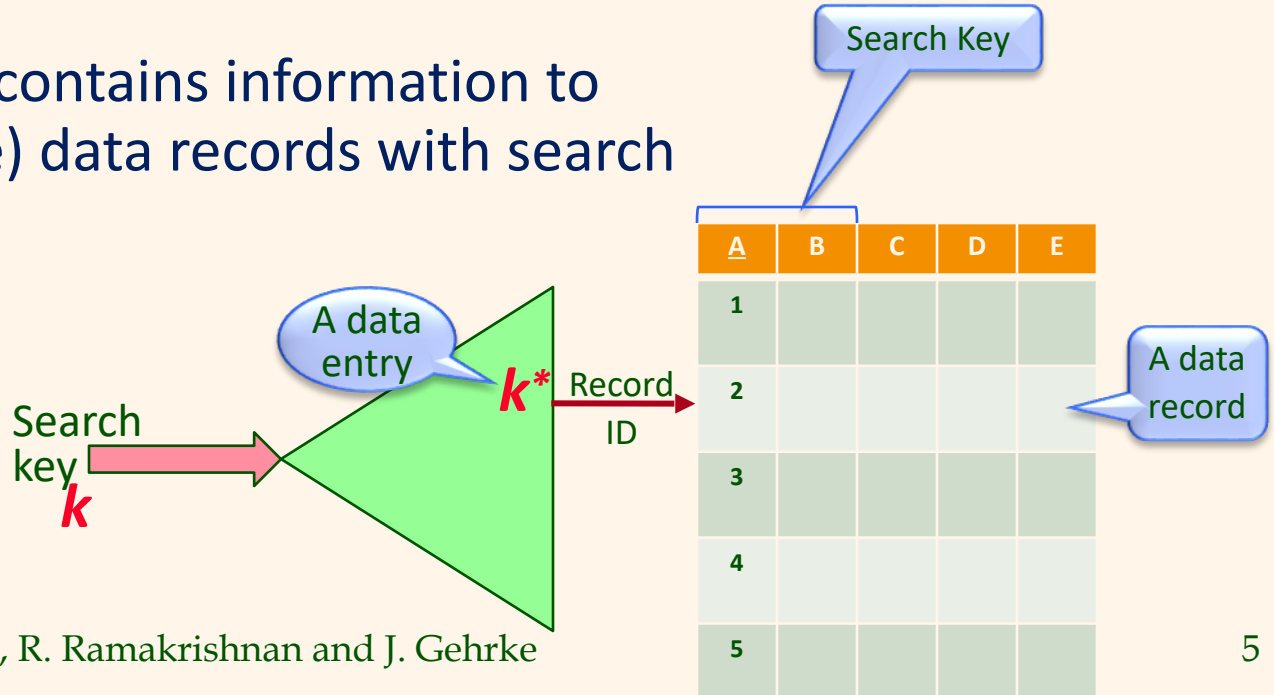
Indexes – Data Entries



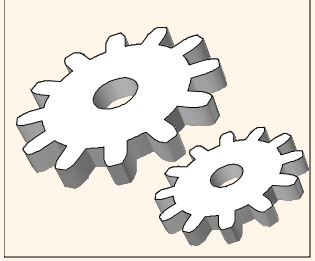
An index contains a collection of *data entries*, and supports efficient retrieval of all data entries ***k**** with a given key value ***k***.

- To locate (one or more) data records with search key value *k*
  - Search the index using *k* to find the desired data entry *k**
  - The data entry *k** contains information to locate (one or more) data records with search key value *k*

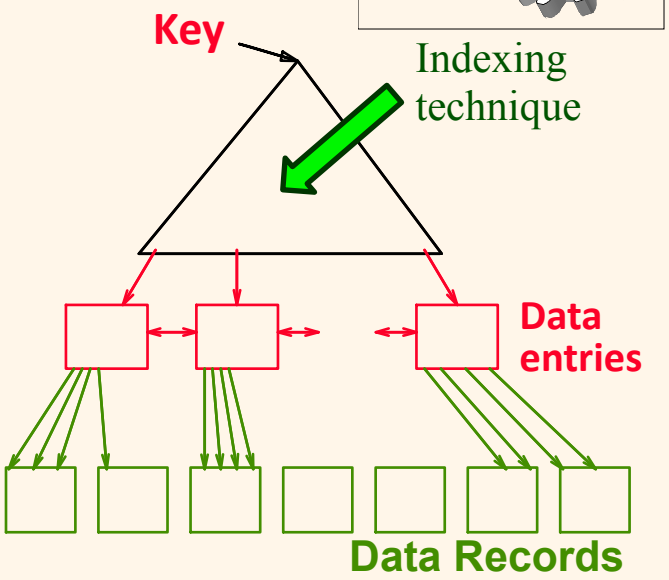




*Alternatives for Data Entry **k*** in Index*

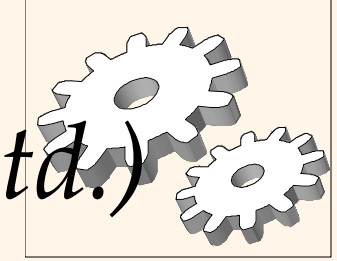


- Three alternatives:
  1. Actual data record (with search key value ***k***)
  2. <***k***, *rid*> pair, where *rid* is the record id of data record with search key value ***k***
  3. <***k***, *rid-list*> pair, where *rid-list* is a list of rids of data records with search key ***k***

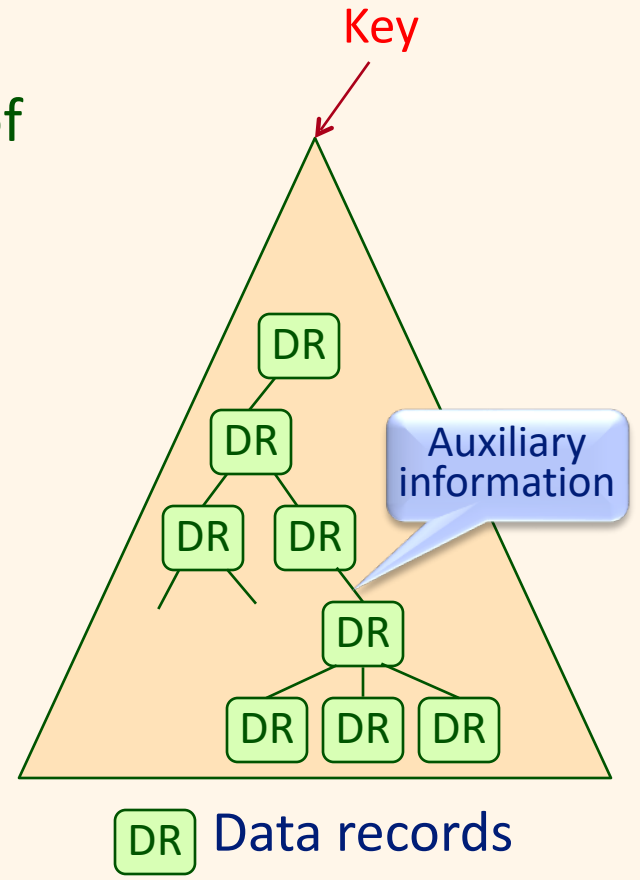


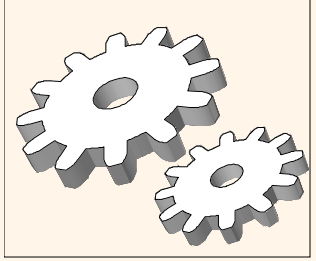
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value ***k***.
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains **auxiliary information** that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)



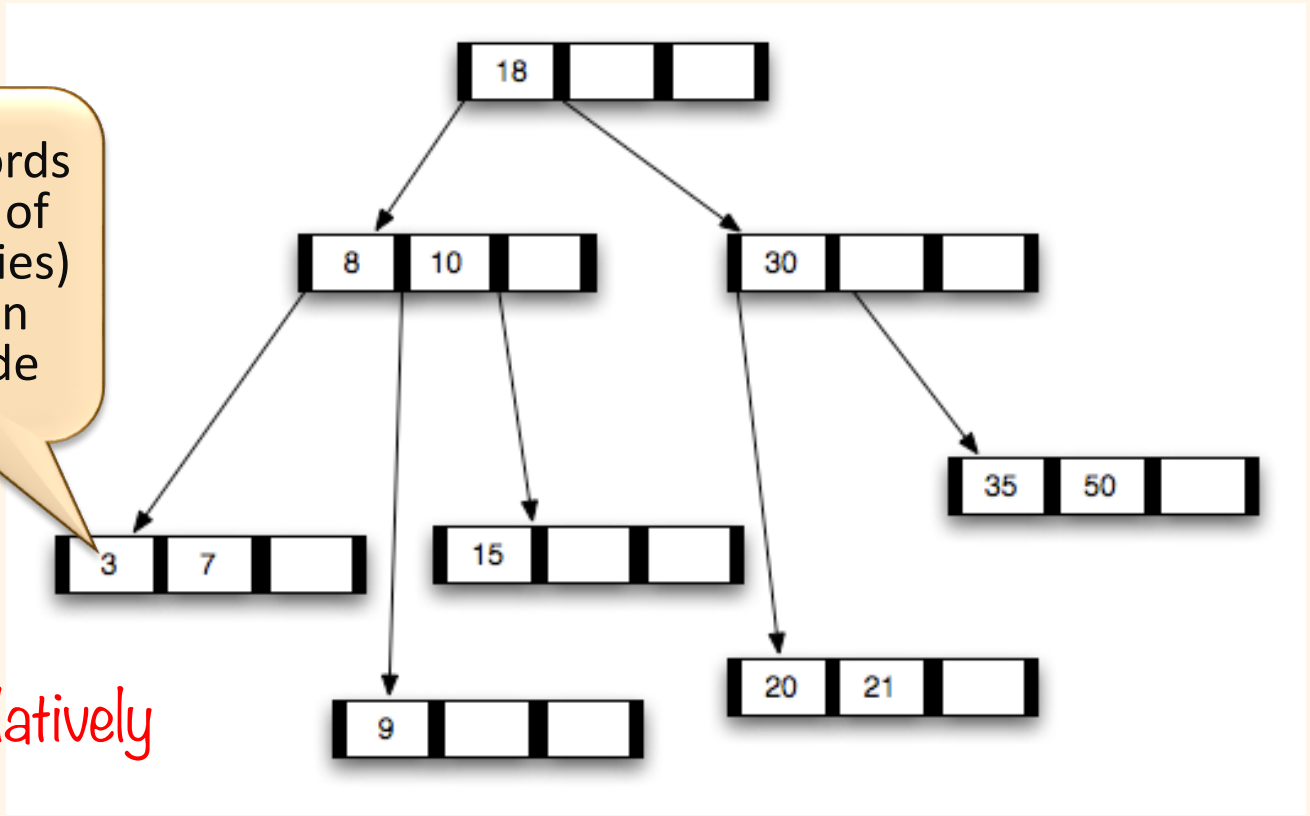
- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
  - If data records are very large, # of pages containing data records is high. Implies size of auxiliary information in the index is also large, typically.





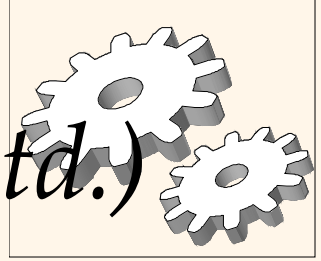
B-tree

B-tree can be used to implement Alternative 1

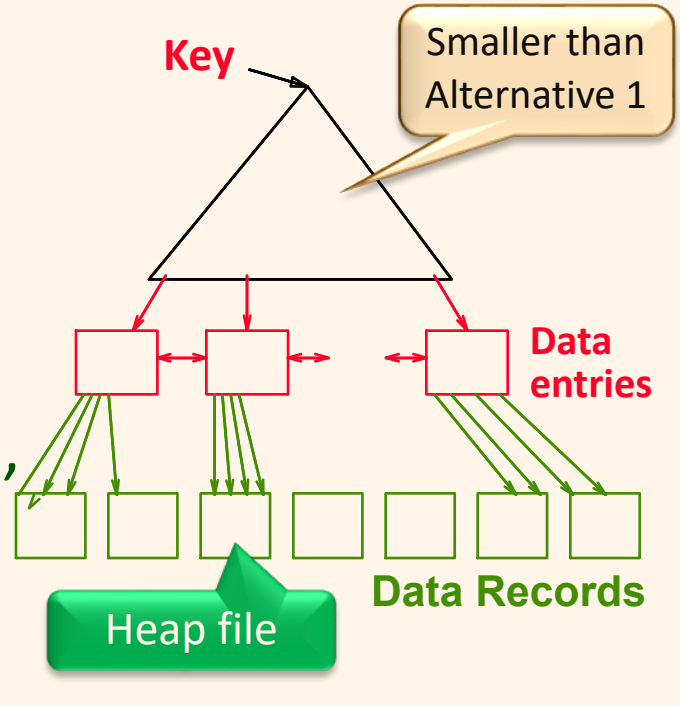


The tree is relatively large

Alternatives for Data Entries (Contd.)

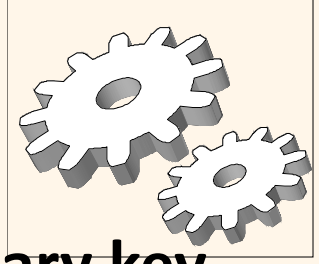


- Alternatives 2: Data entries <***k***, *rid*>, typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)



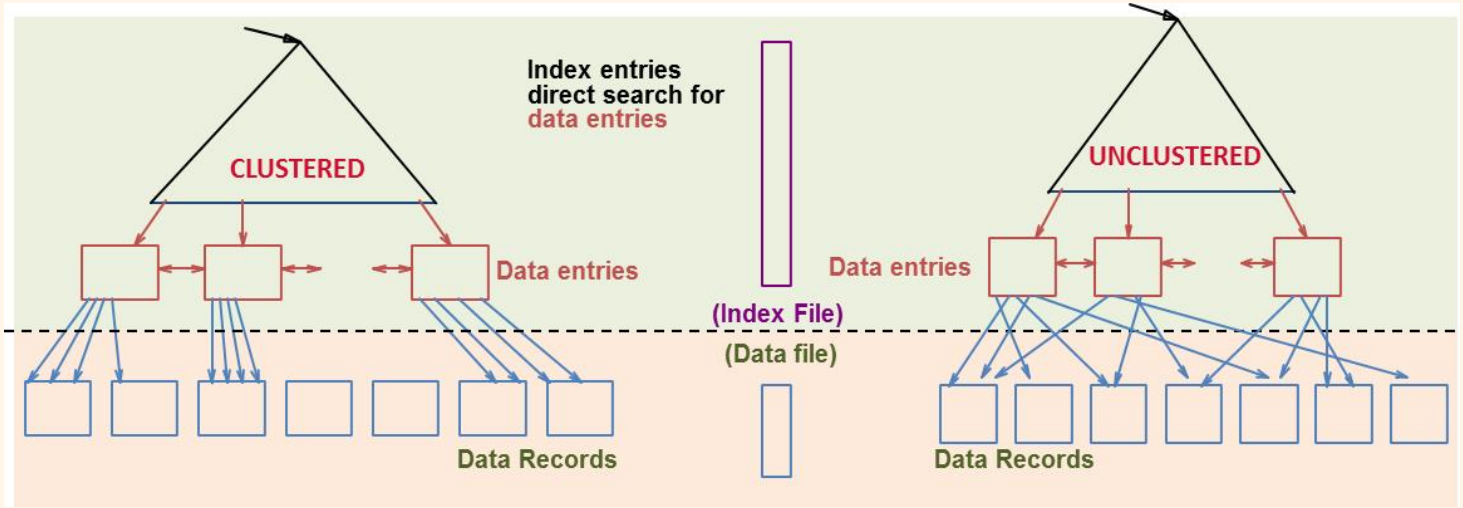
- Alternative 3: Data entries <***k***, list-*rid*>, more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Index Classification

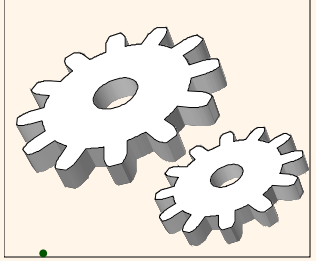


- ***Primary vs. secondary***: If search key contains **primary key**, then called primary index (alternative 1 is usually used to avoid one more I/O to bring in the matching data record).
  - *Unique* index: Search key contains a **candidate key**.
- ***Clustered vs. unclustered***: If **order** of data records is the same as, or `close to', order of data entries, then called clustered index.

Cost of retrieving data records through index varies greatly based on whether index is clustered or not!

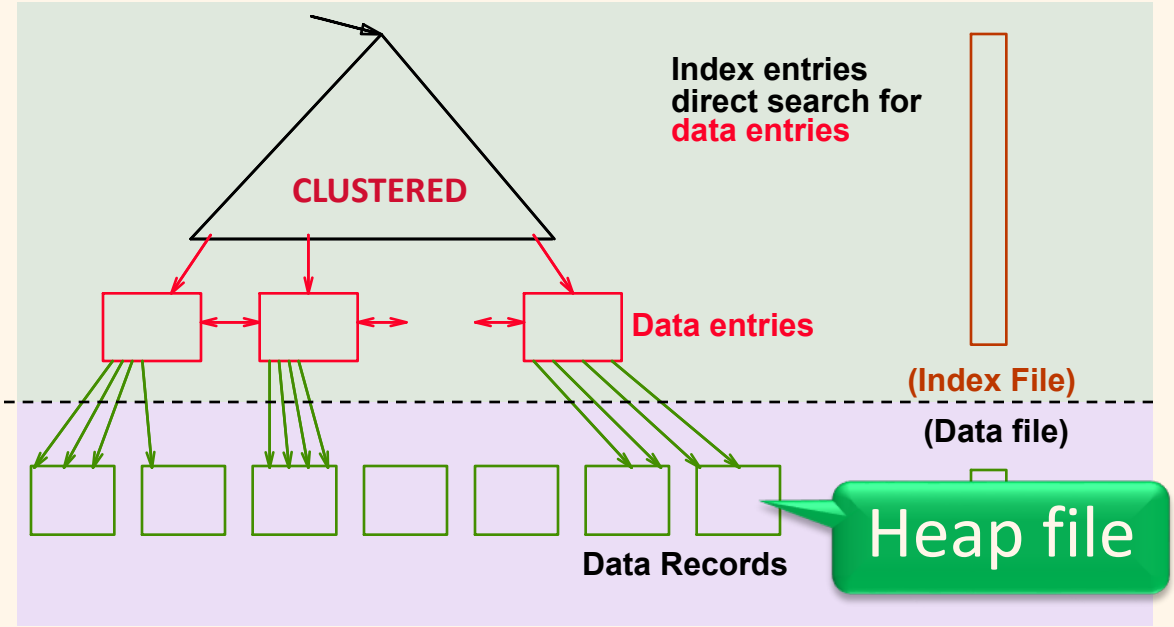


Clustered Index

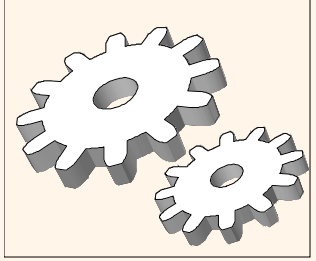


Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.

- To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
- Overflow pages may be needed for inserts. (Thus, order of data records is `close to', but not identical to, the sort order.)

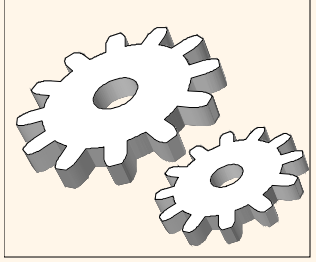


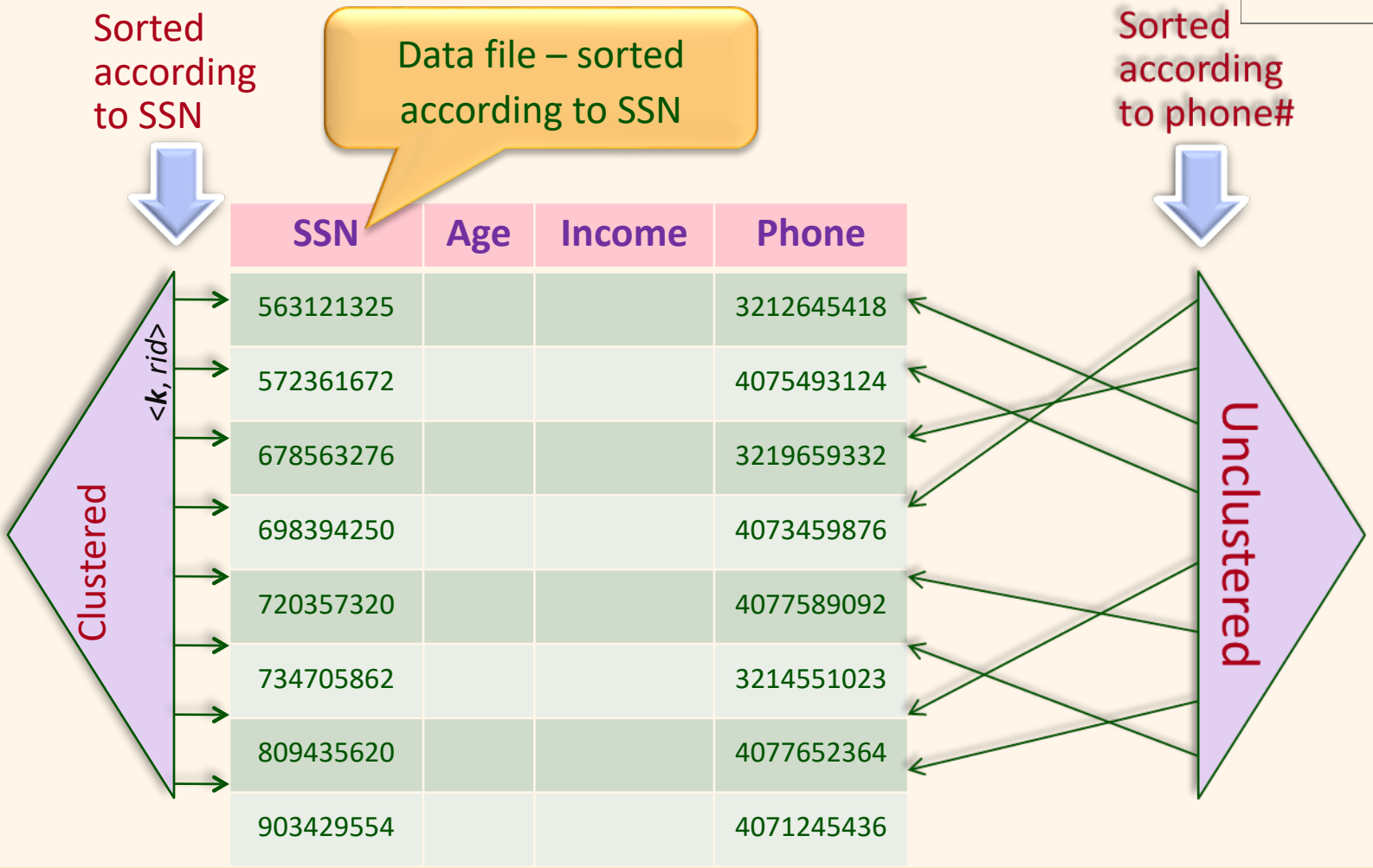
Clustered vs Unclustered



- Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).
- A file can be clustered on at most one search key (more in next page)

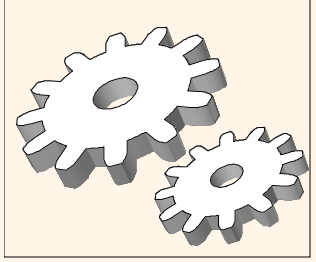
Only One Clustered Index

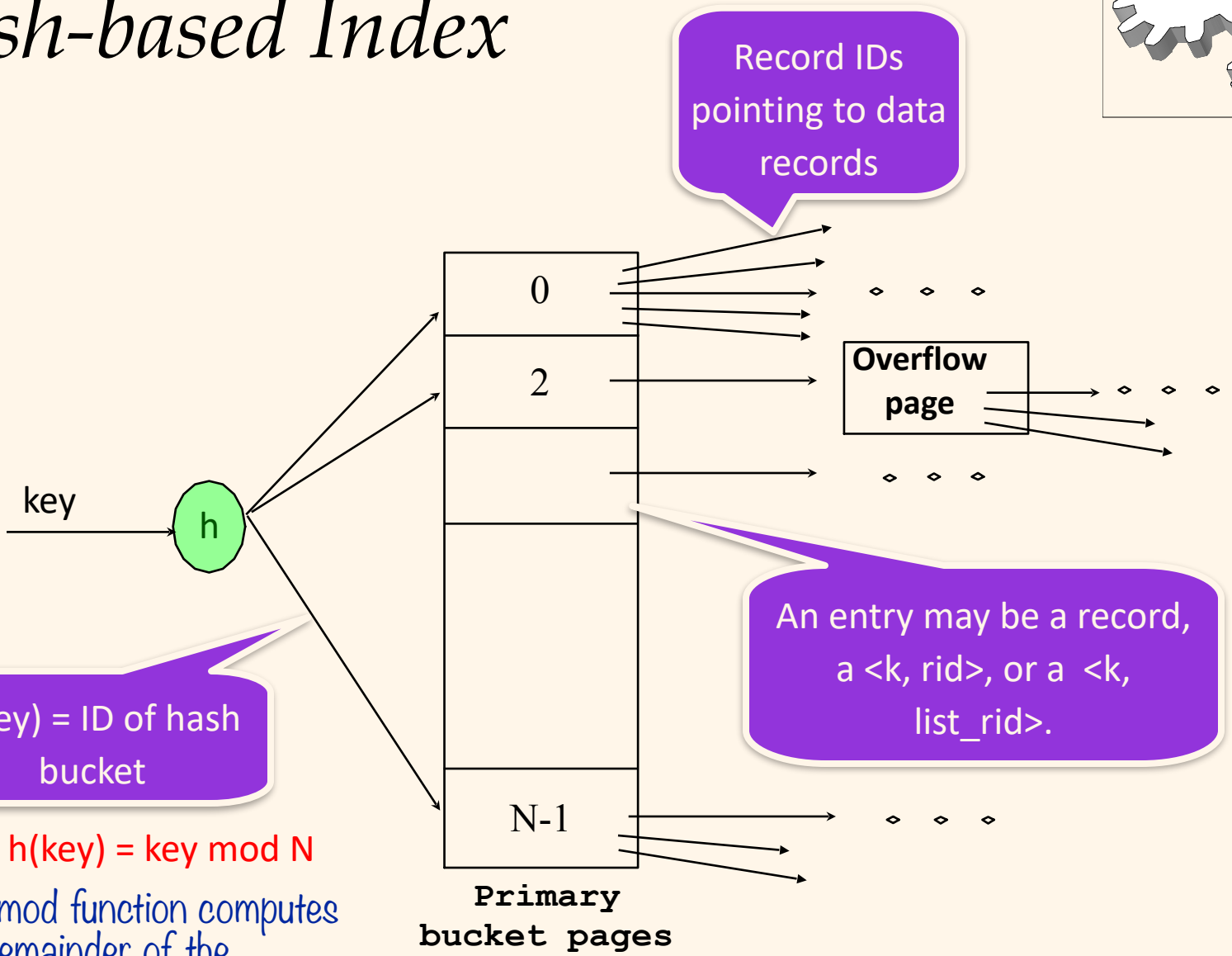




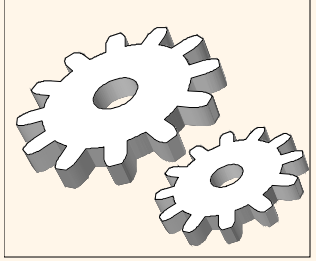
A file can have only one clustered index & alternative 1 often used

Hash-based Index





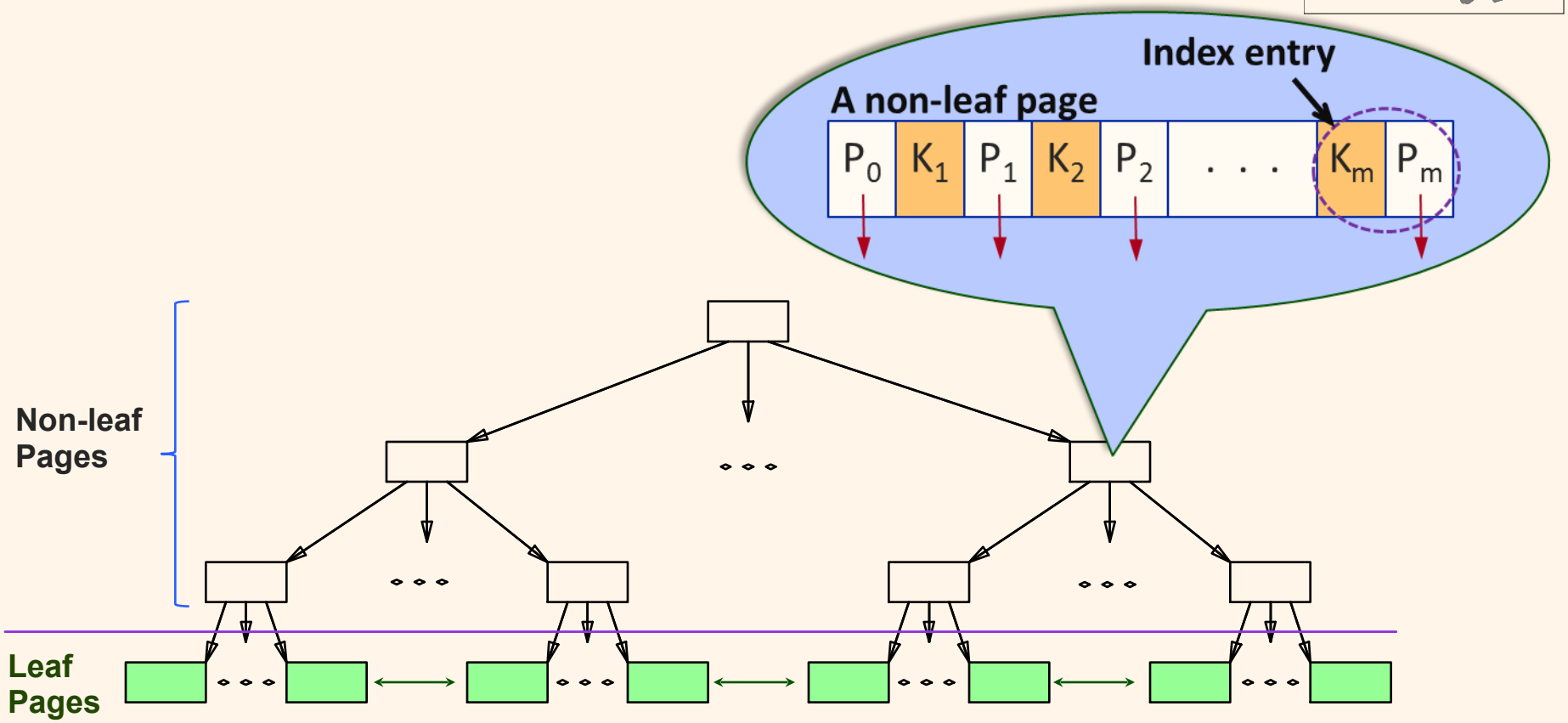
Hash-Based Indexes



Good for equality selections.

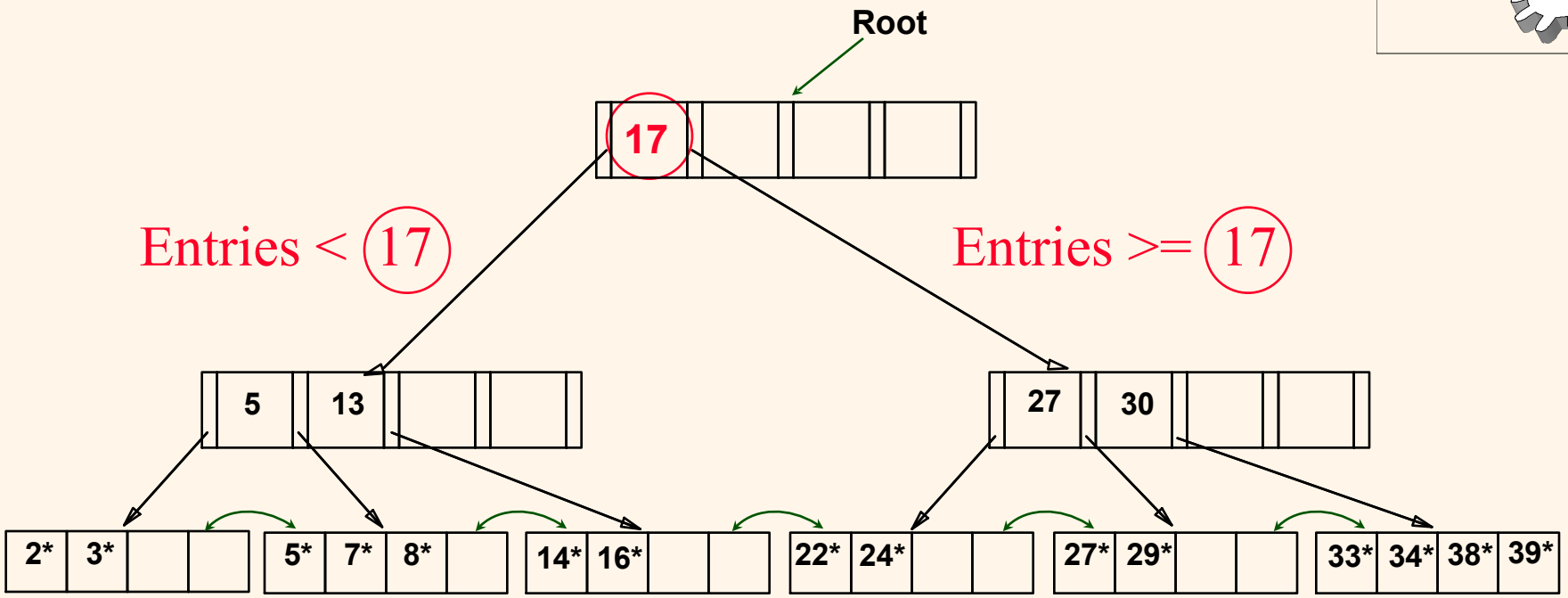
- Index is a collection of *buckets*.
  - Bucket = *primary* page plus zero or more *overflow* pages.
- *Hashing function* **h**: **h** is applied to the *search key* fields of *r*. **h**(*r*) = ID of bucket in which record *r* belongs.
  - Hash on the key fields to determine the bucket(s)
  - Scan the data entries in these buckets to find the matching <*key*, *rid*>
  - Use *rid* to locate the record *r*
- If Alternative (1) is used, the buckets contain the data records (instead of <*key*, *rid*> or <*key*, *rid-list*> pairs)

B+ Tree Indexes: Non-leaf Page



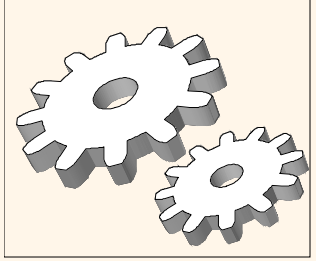
- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages contain *index entries* and direct searches

Example B+ Tree



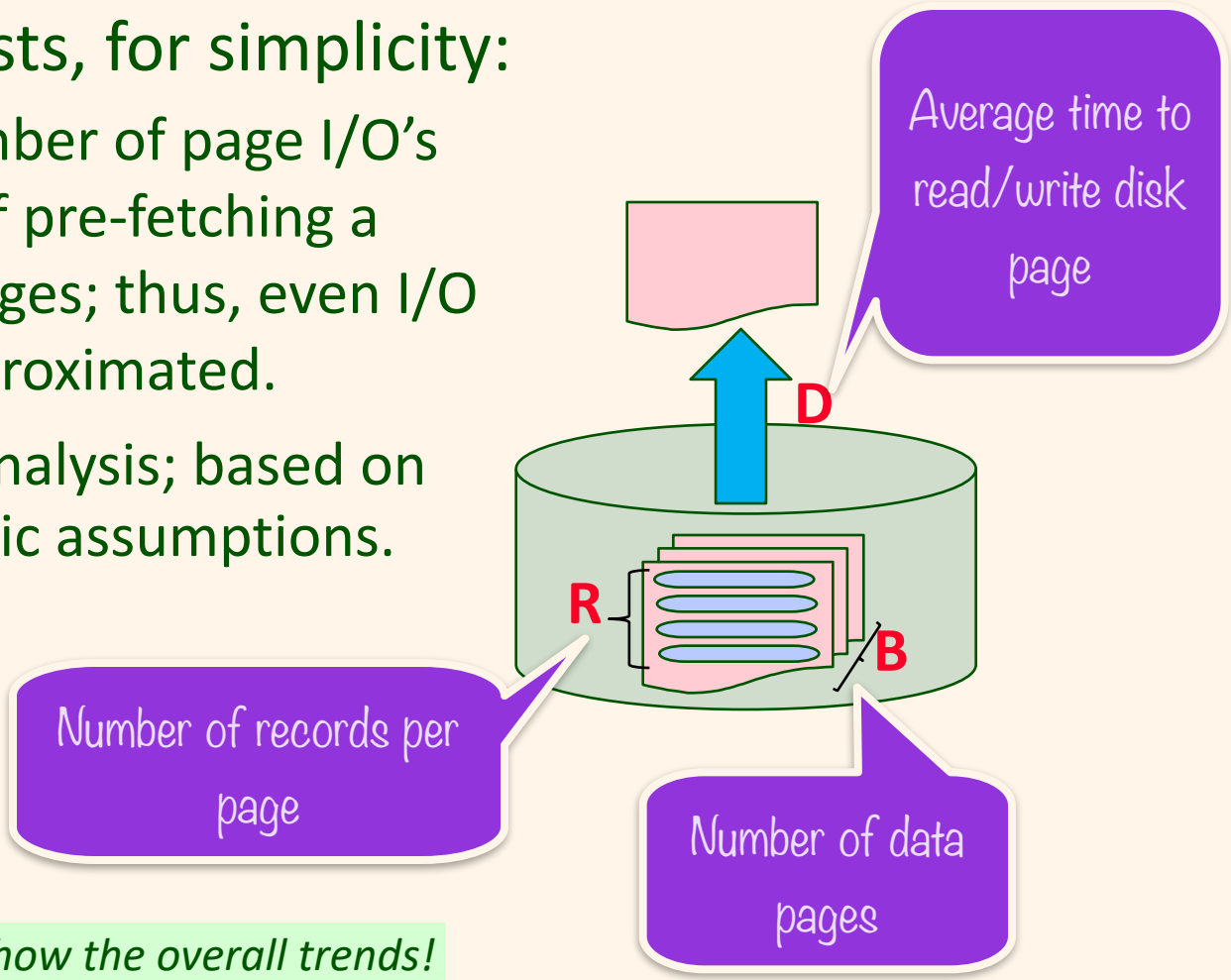
- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

Cost Model for Our Analysis



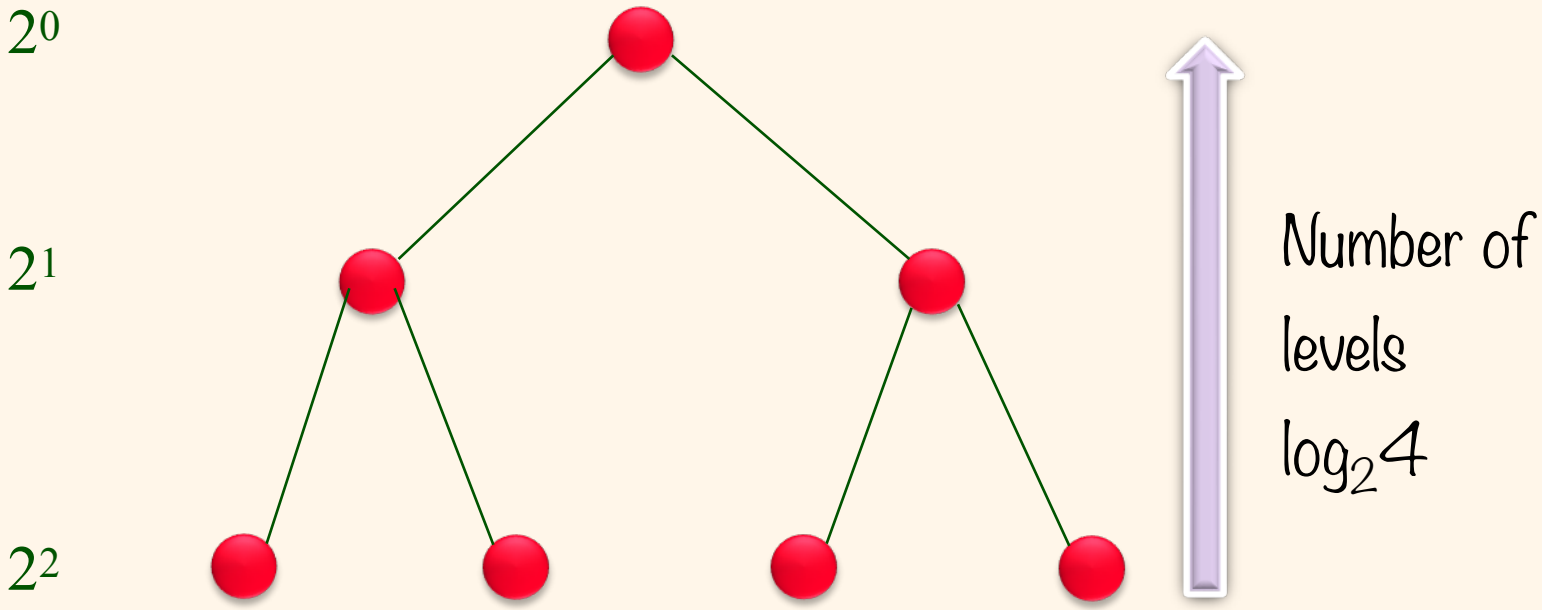
We ignore CPU costs, for simplicity:

- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.



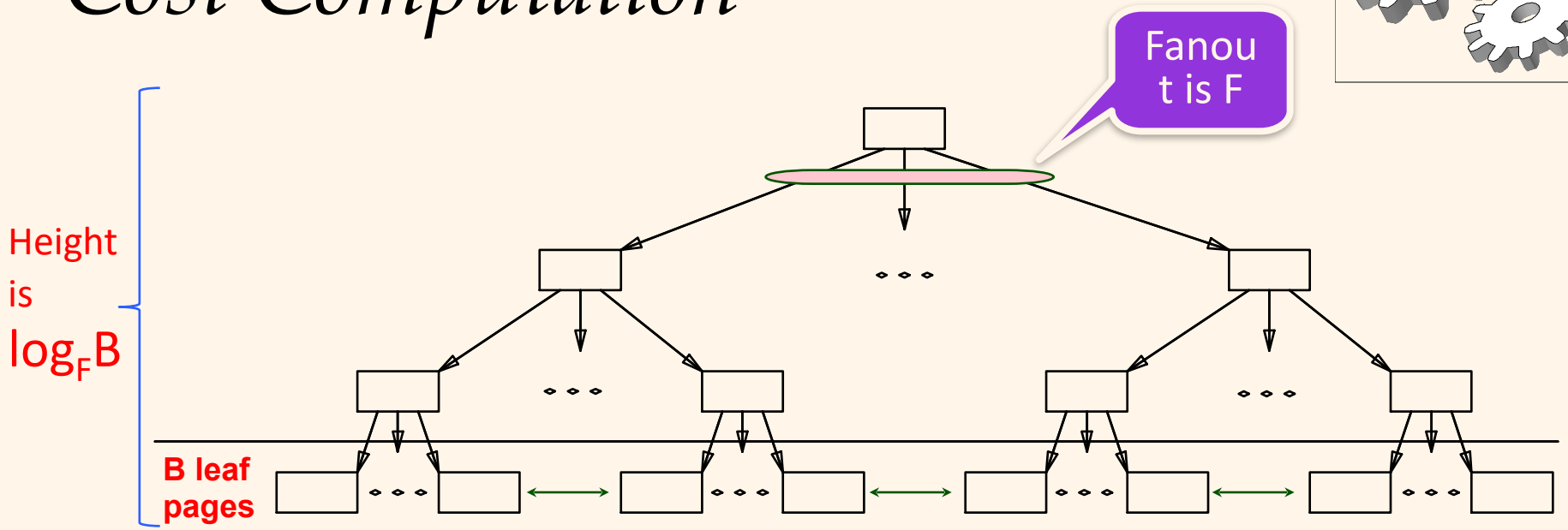
Good enough to show the overall trends!

Review



N leaf nodes with fanout F → $\log_F N$

Cost Computation



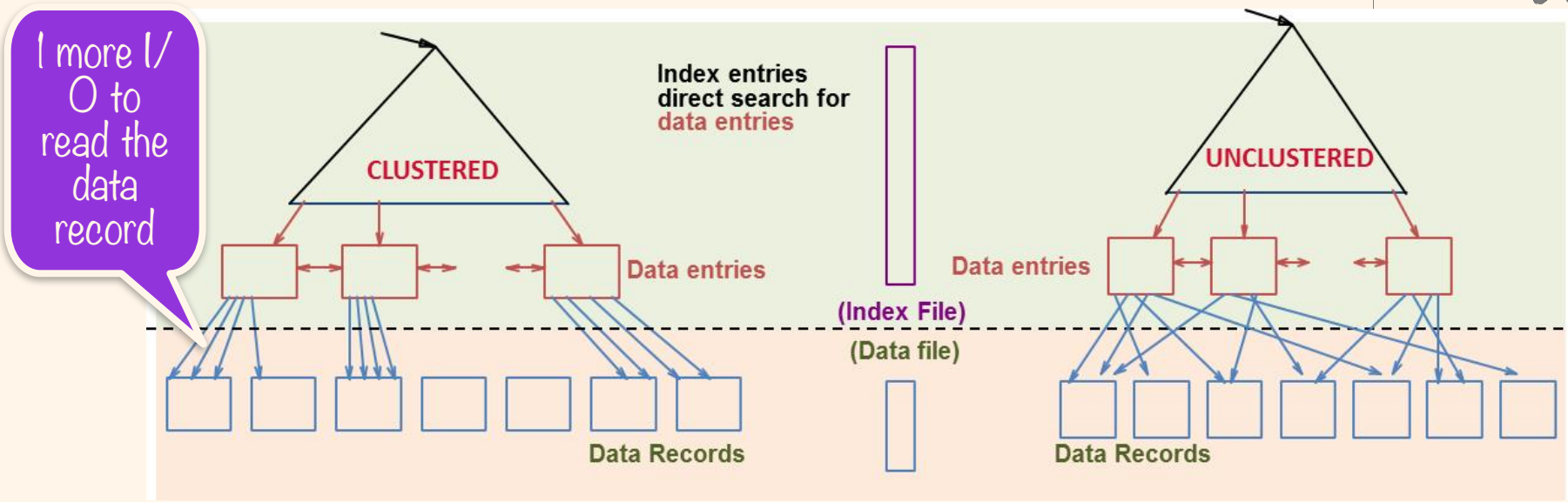
The I/O cost for finding a particular range of 10 records:

- Clustered Index: $D \cdot (\log_F B + 1)$ /* 10 records fit on one page

D is time to read or write a disk page

Time to descend the tree

Cost Computation

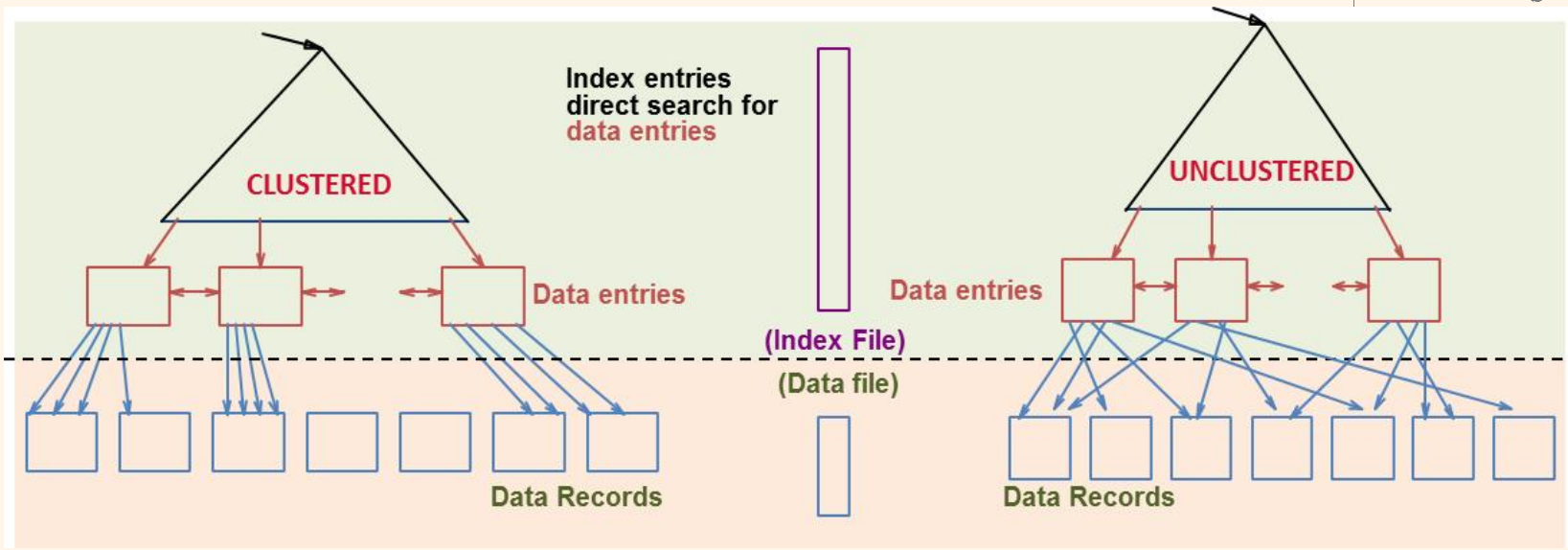


The I/O cost for finding a particular range of 10 records:

- Clustered Index: $D \cdot (\log_F B + 1)$ /* 10 records fit on one page

1 more I/O to read the corresponding data record

Cost Computation



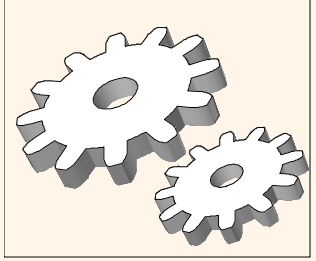
The I/O cost for finding a particular range of 10 records:

- Clustered Index: $D \cdot (\log_F B + 1)$ /* 10 records fit on one page
- Unclustered Index: $D \cdot (\log_F B + 10)$ /* 10 records scattered over different pages

Cost of retrieving data records through index varies greatly based on whether index is clustered or not!

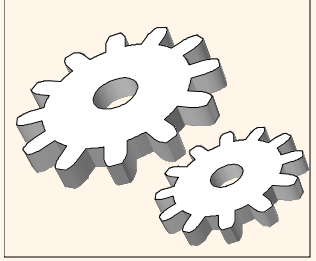
Fetch 10 data pages

Comparing File Organizations

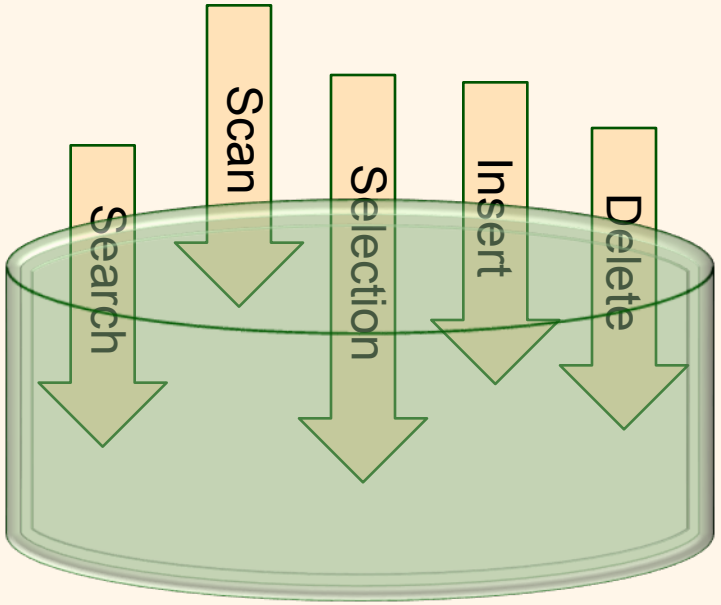


1. Heap files (random order; insert at eof)
2. Sorted files, sorted on ***<age, sal>***
3. Clustered B^+ tree file, Alternative (1), search key ***<age, sal>***
4. Heap file with unclustered B^+ tree index on search key ***<age, sal>***
5. Heap file with unclustered hash index on search key ***<age, sal>***

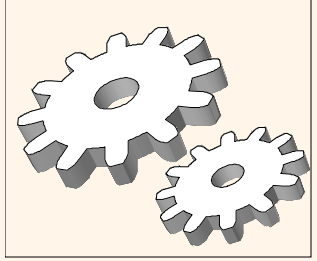
Operations to Compare



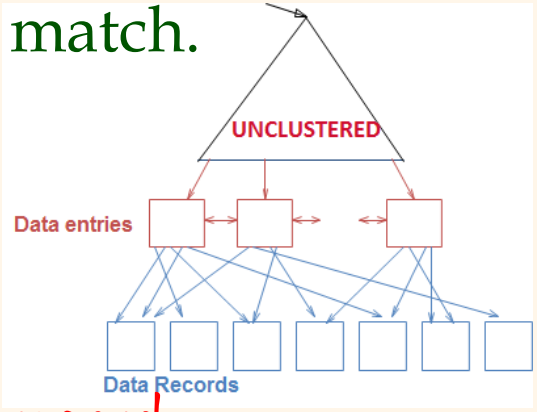
- Scan: Fetch all records from disk
- Equality search
- Range selection
- Insert a record
- Delete a record



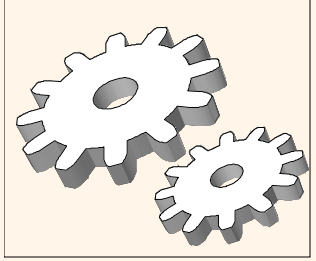
Assumptions in Our Analysis



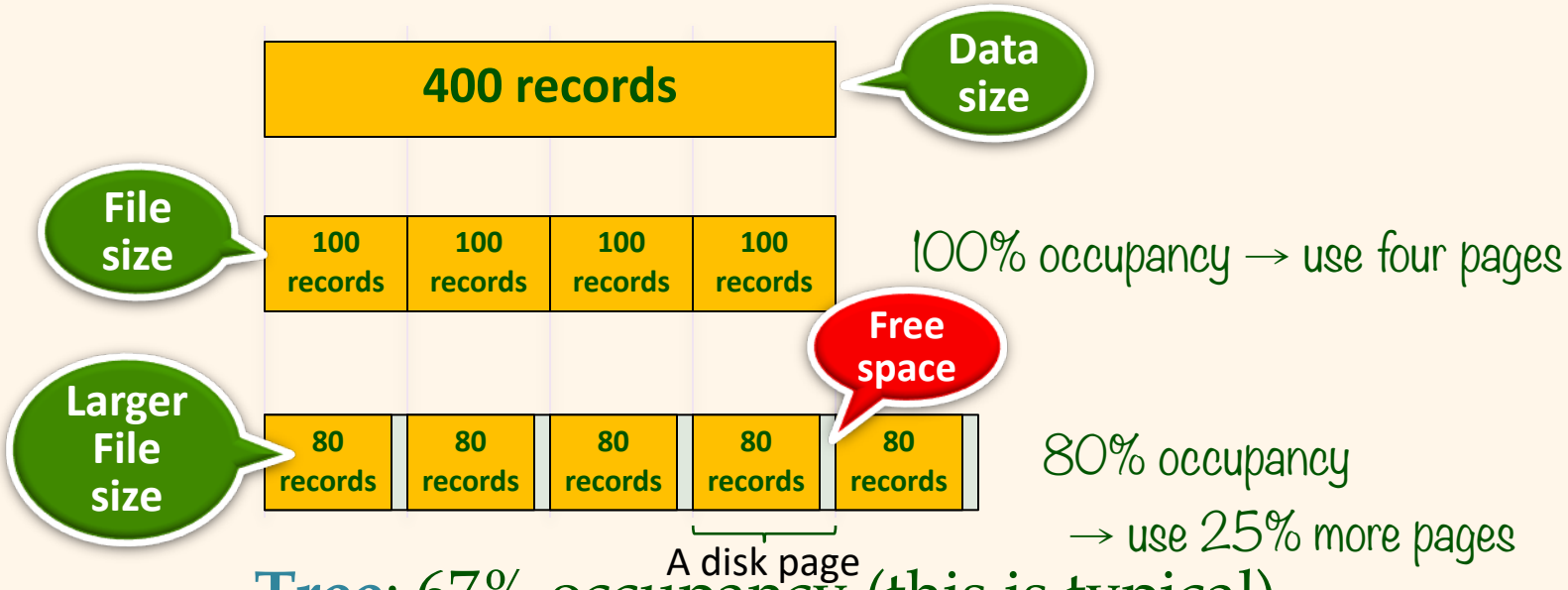
- Heap Files:
  - Equality selection on key; exactly one match.
- Sorted Files:
  - Files compacted after deletions.
- Indexes:
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - **80% page occupancy** → File size = 1.25 data size (next page)
  - Tree: 67% occupancy (this is typical).
    - **Implies** file size = 1.5 data size (next page)



Assumptions in Our Analysis



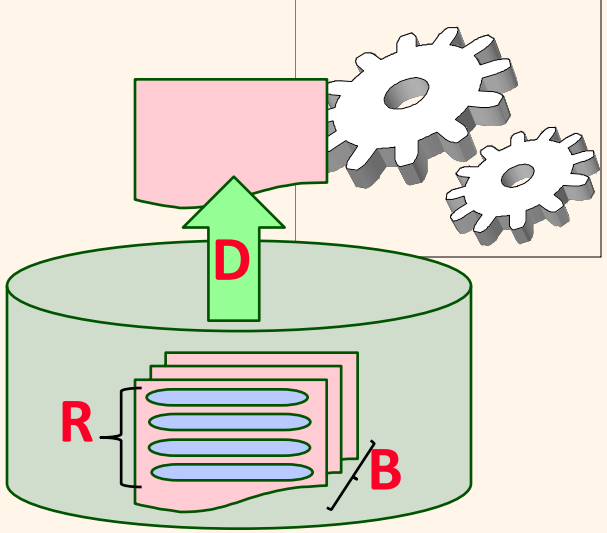
- **Hash**: No overflow buckets.
  - **80% page occupancy** → File size = 1.25 data size



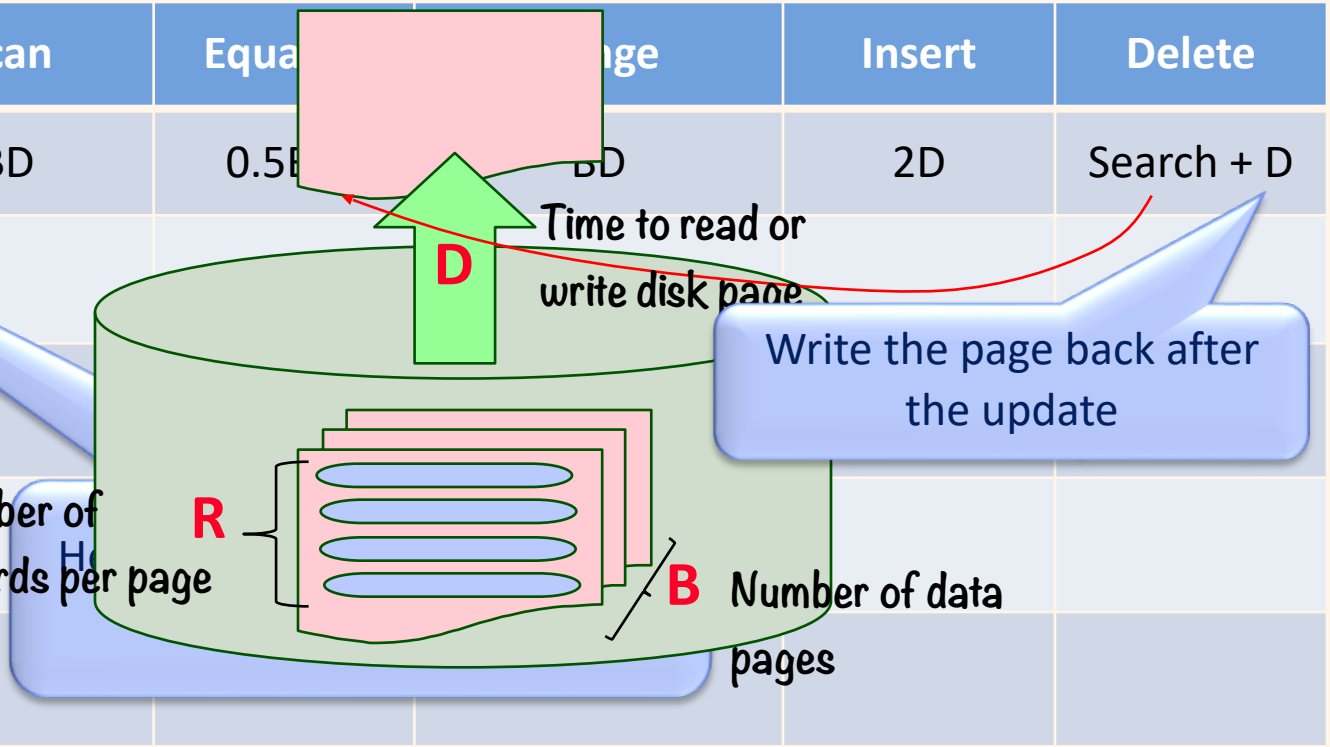
- **Tree**: 67% occupancy (this is typical).
  - **Implies** file size = 1.5 data size

Cost of Operations

Several assumptions underline these (rough) estimates!

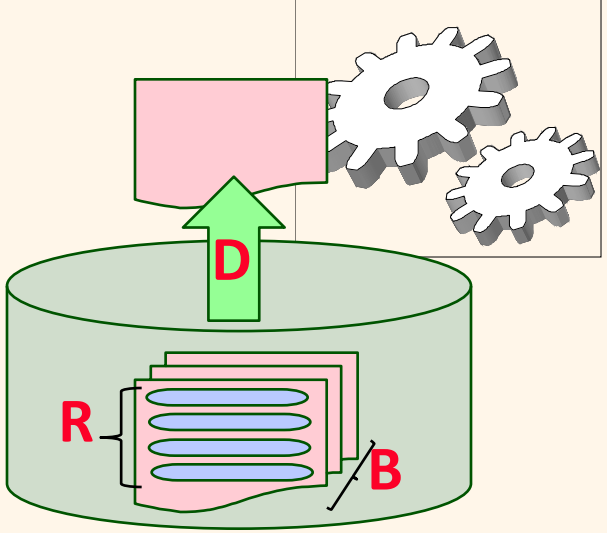


	Scan	Equa	ge	Insert	Delete
Heap	BD	0.5E	BD	2D	Search + D
Sorted					
Clustered					
Unclustered Tree Index					
Unclustered Hash Index					



Cost of Operations

Several assumptions underlie these (rough) estimates!



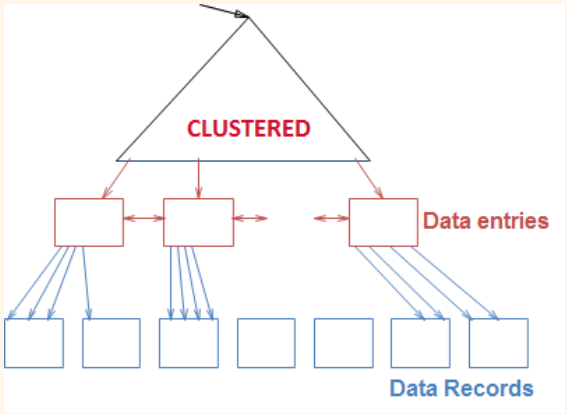
	Scan	Equality	Range	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search + D
Sorted	BD	$D \cdot \log_2 B$	$D(\log_2 B$ + # matching pages)	Search + BD	Search + BD
Clustered					
Unclustered Tree Index					
Unclustered Hash Index					

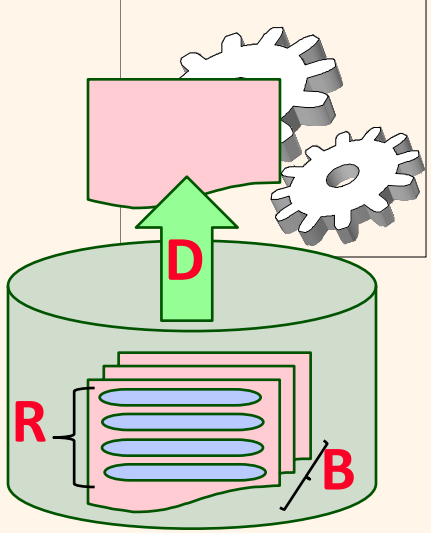
Sorted files, sorted on <age, sal>

Fetch & rewrite the latter half of the file after adding the new record

Cost of Operations

Several assumptions underlie these (rough) estimates!



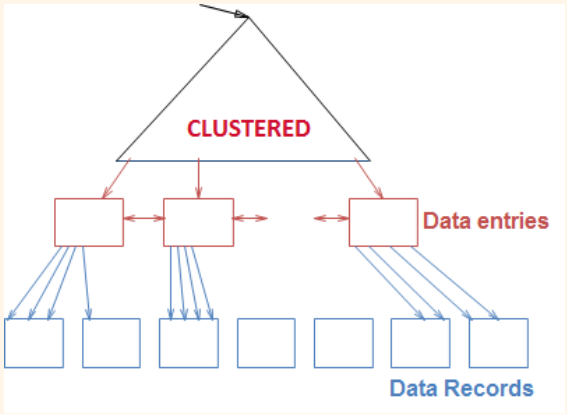


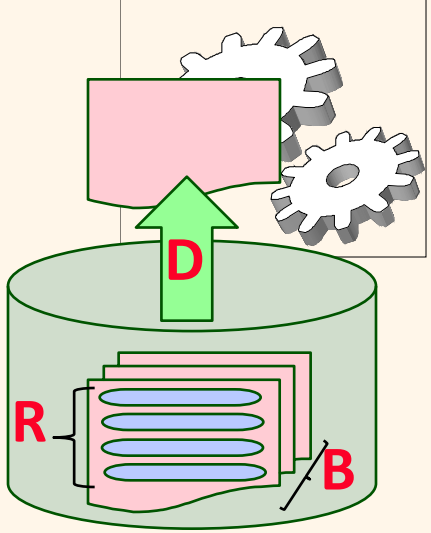
	Scan	Equality	Range	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search + D
Sorted	BD	$D \cdot \log_2 B$	$D \cdot (\log_2 B$ + # matching pages)	Search + BD	Search + BD
Clustered					
Unclustered Tree Index					
Unclustered Hash Index					

Clustered B+ tree file, Alternative (1), search key <age,sal>

Cost of Operations

Several assumptions underlie these (rough) estimates!





	Scan	Equality	Range	Insert	Delete
	BD	.5BD	BD	2D	Search + D
Sorted	BD	D·log	og$_2$B + # matching pages)	Search + BD	Search + BD
Clustered	1.5BD	D·log$_F$(1.5B)	D·(log$_F$(1.5B) + # matching pages)	Search + D	Search + D
Unclustered Tree Index					
Unclustered Hash Index					

67% page occupancy, 50% more pages to scan

Height of the tree

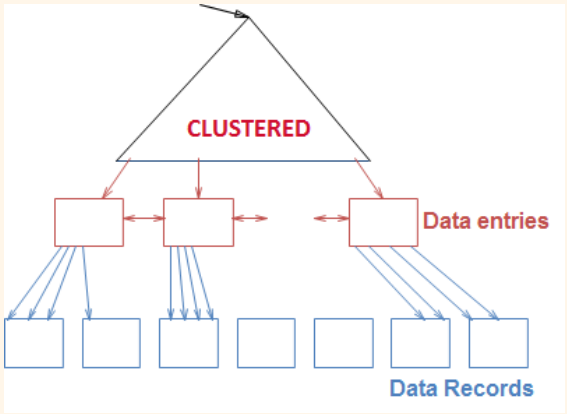
Number of pages as leaf nodes

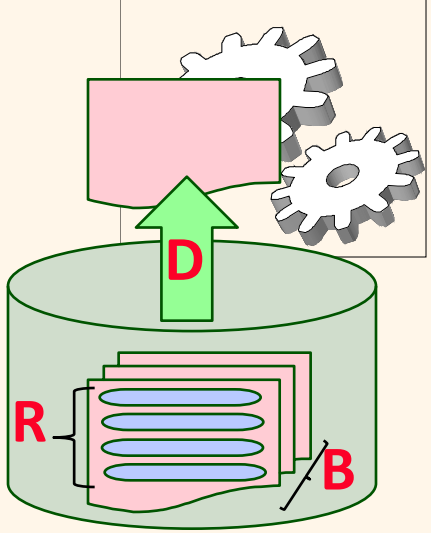
Clustered B+ tree file, Alternative (1), search key <age,sal>

1 write to insert the new record

Cost of Operations

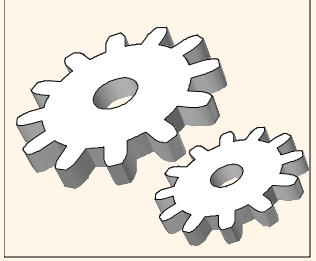
Several assumptions underlie these (rough) estimates!





	Scan	Equality	Range	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search + D
Sorted	BD	$D \cdot \log_2 B$	$D \cdot (\log_2 B$ + # matching pages)	Search + BD	Search + BD
Clustered	1.5BD	$D \cdot \log_F (1.5B)$	$D \cdot (\log_F (1.5B)$ + # matching pages)	Search + D	Search + D
Unclustered Tree Index	BD(R+0.15)	$D(1 + \log_F (1.5B))$	$D \cdot (\log_F (1.5B)$ + # matching pages)	$D(3 + \log_F (1.5B))$	Search + 2D
Unclustered Hash Index	BD(R+0.125)	2D	BD	4D	Search + 2D

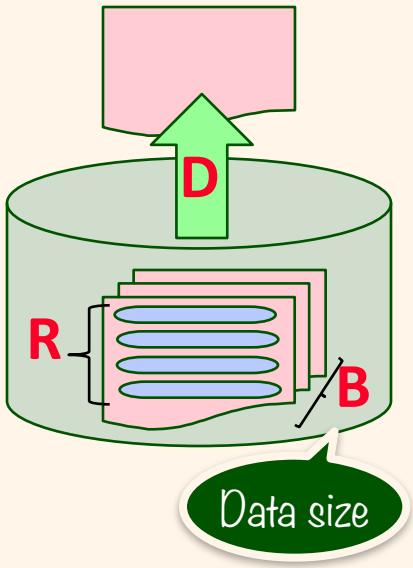
Cost of Operations
Heap File /w Unclustered B+ tree



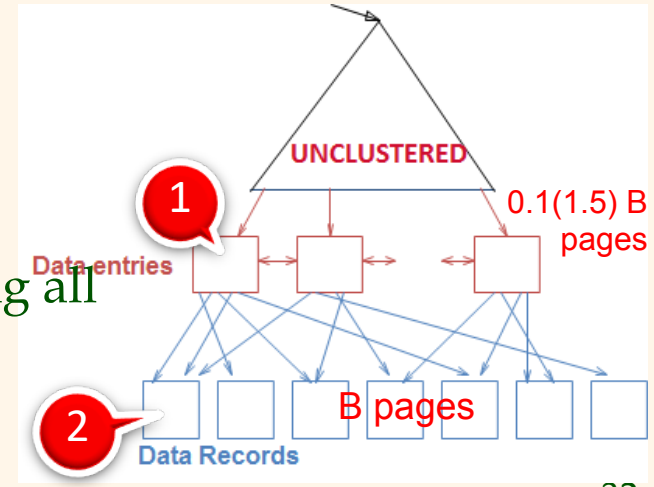
SCAN (to obtain data records in sorting order)

- Scan the leaf level of the index (1)
- For each data entry in a leaf node, fetch the corresponding data record from the heap file (2)

SCAN COST: **0.1(1.5B)D + BRD** = BD(R+0.15)

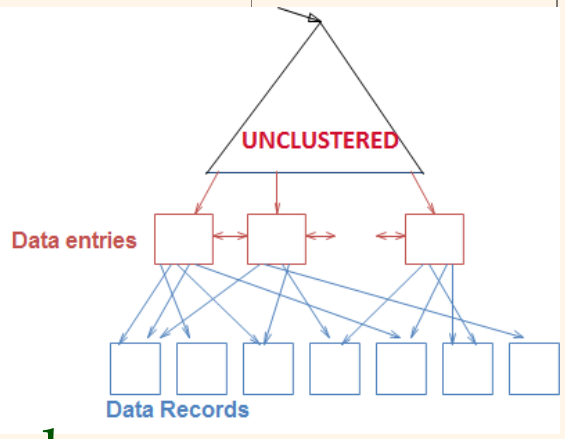


- **Cost of scanning the leaf nodes**
  - Each page is 67% occupied ⇒ **# data pages is 1.5B**
  - Data entry is only 10% the size of data record ⇒ **# leaf pages is 0.1(1.5B)**
  - Cost of scanning the leaf pages is **0.1(1.5B)·D**
- **Cost of fetching the data records**
  - Number of data record is B·R Cost of retrieving all data records is **BR·D**



Cost of Operations

Heap File /w Unclustered B+ tree



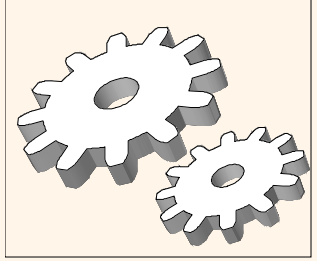
EQUALITY SEACH

- Search for the matching data entry in the index
- Fetch the corresponding data record from the data file

SEARCH COST:

- Cost of searching the index
  - # leaf pages is 0.1(1.5B) ⇒ tree height is $\log_F(0.15B)$
  - Descending the index tree visits $\log_F(0.15B)$ pages
  - Cost of finding the matching data entry is **$D \cdot \log_F(0.15B)$**
- Cost of fetching the data records
  - Fetching the corresponding data records incurs one more I/O, or **$1 \cdot D$**
- Total search cost: **$D \cdot \log_F(0.15B) + 1D = D(1+ \log_F(0.15B))$**

Cost of Operations
Heap File /w Unclustered B+ tree

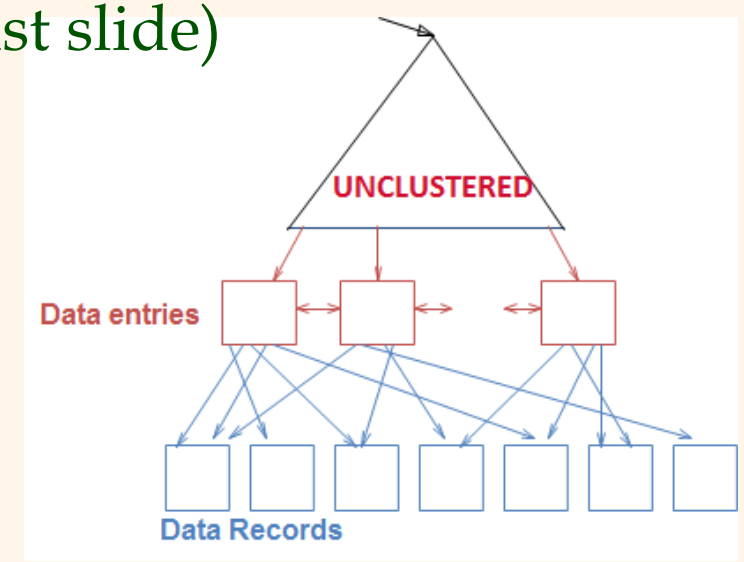


- Equality Search (from last slide)

$$D(1 + \log_F(0.15B))$$

Fetching the matching record

Search the B+ tree



- Range Selection

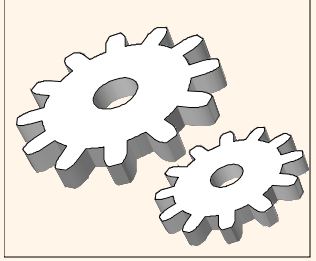
$$D(\#\ matches + \log_F(0.15B))$$

Fetching each match in the range incurs one I/O

Search the B+ tree

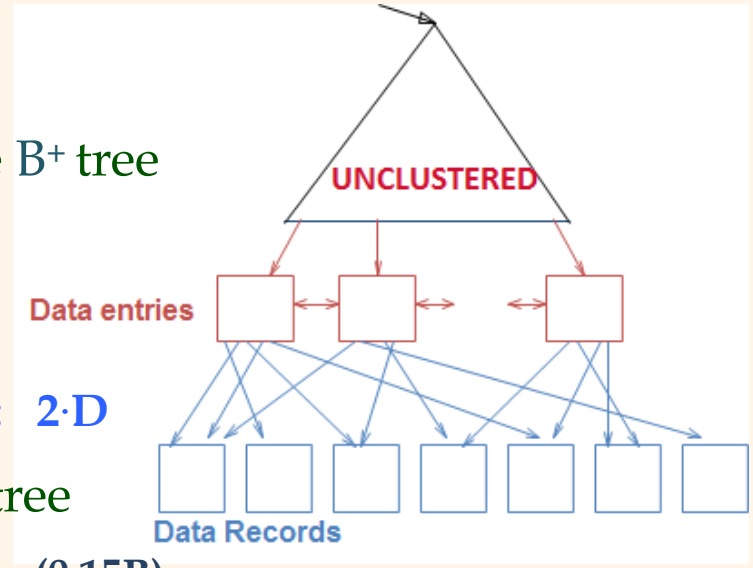
Cost of Operations

Heap File /w Unclustered B+ tree



INSERT

- Insert the new record in the heap file
- Insert the corresponding data entry in the B+ tree

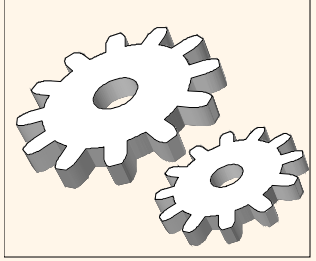


INSERT COST:

- Cost of inserting the new record
  - Inserting the new record incurs two I/O's: **2·D**
- Cost of inserting the data entry in the B+ tree
  - # leaf pages is 0.1(1.5B) ⇒ tree height is $\log_F(0.15B)$
  - Descending the index tree visits $\log_F(0.15B)$ pages
  - Cost of finding the target leaf page is $D \cdot \log_F(0.15B)$
  - Updating target leaf page incurs one more I/O: $1D + D \cdot \log_F(0.15B)$
- Total insert cost:

$$D + D \cdot \log_F(0.15B) + 2D = D(3 + \log_F(0.15B))$$

Cost of Operations
Heap File /w Unclustered B+ tree

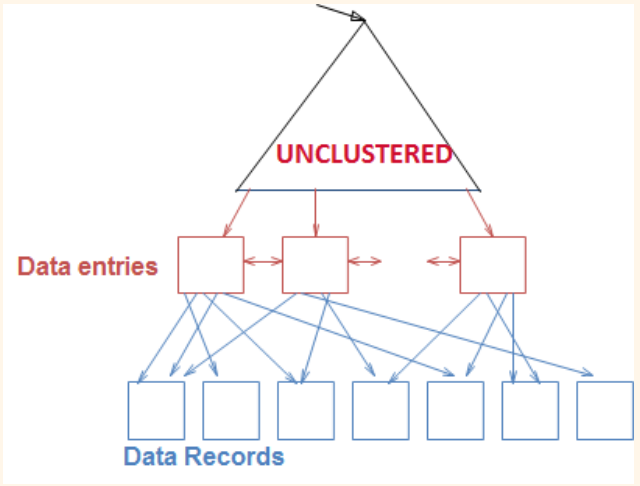


- **Insert** (from last slide)

$$D(3 + \log_F(0.15B))$$

1 I/O to insert the data entry + 2 I/O's to insert the new record

Search the B+ tree



- **Delete**

$$D(3 + \log_F(0.15B)) = 2D + \text{Search}$$

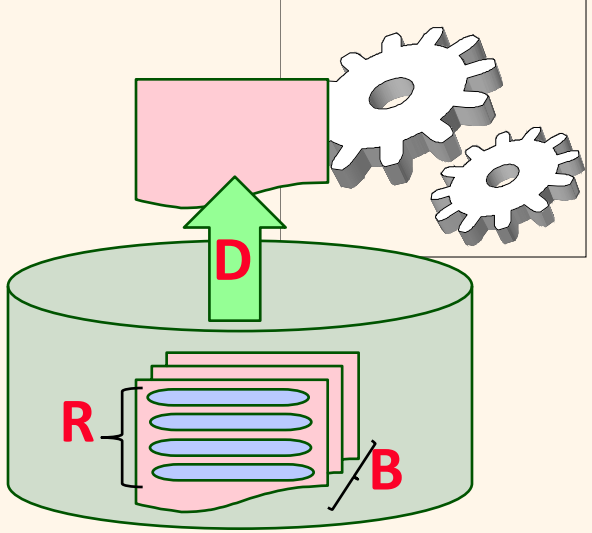
1 I/O to delete the data entry + 2 I/O's to delete the data record

Search the B+ tree

1 I/O to write back the data-entry page and another I/O to write back the data-record page

Cost of Operations

Several assumptions underlie these (rough) estimates!



	Scan	Equality	Range	Insert	Delete
Heap					Search + D
Sorted					
Clustered		$D \cdot \log_F 1.5B$	$D \cdot (\log_F 1.5B$ + # matching pages)		
Unclustered Tree Index	$BD(R+0.15)$	$D(1 + \log_F 0.15B)$	$D \cdot (\log_F 0.15B$ + # matches)	$D(3 + \log_F 0.15B)$	Search + 2D
U… H…					

Heap file with unclustered B+ tree index on search key <age,sal>

1 I/O to insert the data entry + 2 I/O's to insert the data record

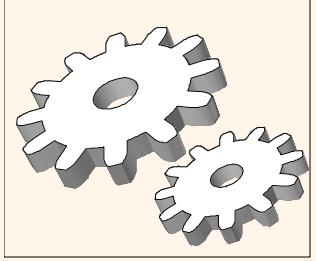
Unclustered ⇒ one I/O per record

Cost of scanning data entries is $0.1(1.5B) \cdot D$

Each match requires an I/O

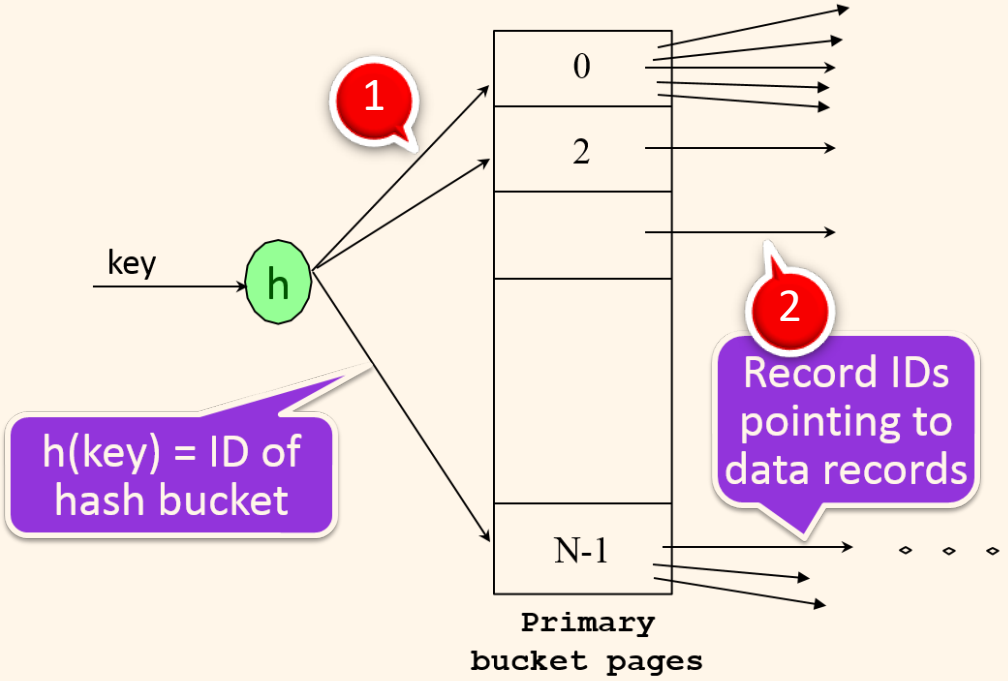
1 I/O's to write back the data-entry page and 1 I/O to write back the data-record page

Cost of Operations (1)
Heap File /w Unclustered Hash Index

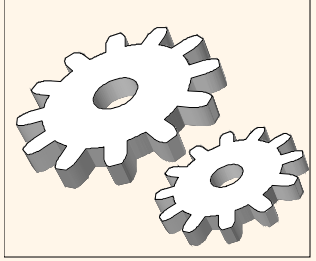


SCAN (to obtain data records in "hash" order)

- Fetch the hash buckets 1
- For each data entry in a hash bucket, fetch the corresponding data record from the heap file 2



Cost of Operations (1)
Heap File /w Unclustered Hash Index



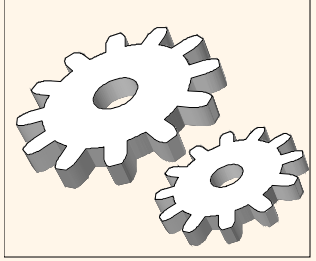
SCAN (to obtain data records in "hash" order)

- Fetch the hash buckets
- For each data entry in a hash bucket, fetch the corresponding data record from the heap file

SCAN COST **0.125BD + BRD = BD(R+0.125)**

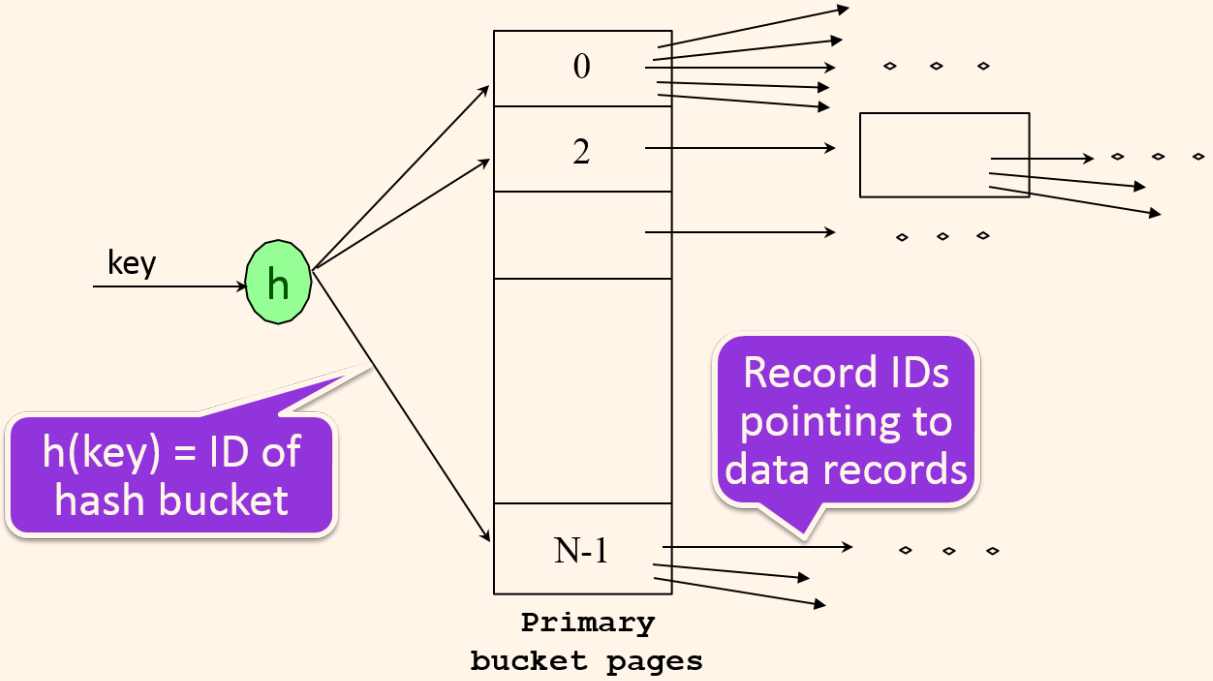
- Cost of scanning the hash buckets
  - Each page is 80% occupied ⇒ # data pages is 1.25B
  - Data entry is only 10% the size of data record ⇒ # index pages (i.e., # hash buckets) is 0.1(1.25B) = 0.125B
  - Cost of scanning the data entry is **0.125B·D**
- Cost of fetching the data records
  - Since number of data record is B·R, cost of retrieving all data records is **BR·D** (i.e., 1 I/O per record)

Cost of Operations (2)
Heap File /w Unclustered Hash Index

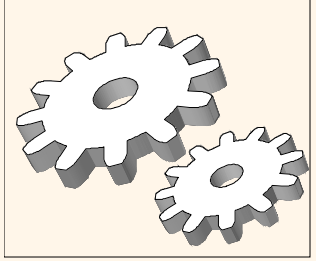


Range Selection (Hash structure cannot help)

- Scan the hash buckets
- For each hash bucket, fetch the data record from the heap file if the corresponding data entry is within the range



Cost of Operations (2)
Heap File /w Unclustered Hash Index



Range Selection (Hash structure cannot help)

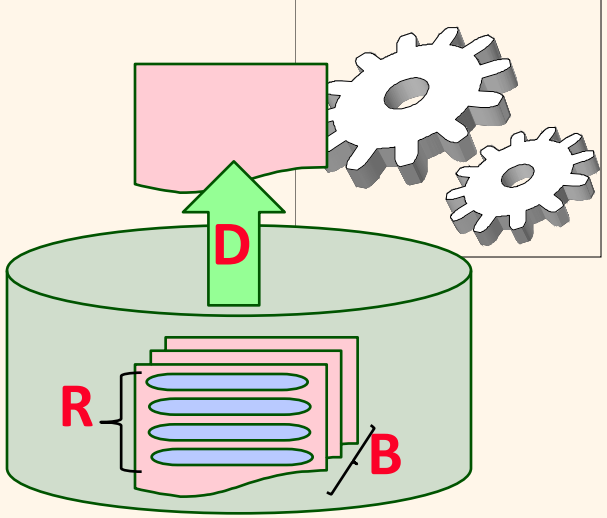
- Scan the hash buckets
- For each hash bucket, fetch the data record from the heap file if the corresponding data entry is within the range

SCAN COST: **0.125BD + (# matches) D = D·(0.125B + #matches)**

- Cost of scanning the hash buckets
  - Each page is 80% occupied ⇒ # data pages is 1.25B
  - Data entry is only 10% the size of data record ⇒ # index pages (i.e., # hash buckets) is 0.1(1.25B) = 0.125B
  - Cost of scanning the data entry is **0.125B·D**
- Cost of fetching the data records:
  - **(# matches)·D**

Cost of Operations

Several assumptions underlie these (rough) estimates!



	Scan	Equality	Range	Insert	Delete
Heap	BD	0.5BD	BD		Search + D
Cluster	1.5BD	D·log_F 1.5B	D·log_F 1.5B + # matches		
Unclustered Tree Index	BD(R+0.15)	D(log_F 0.15B + # matches)	D(log_F 0.15B + # matches)	D(3 + log_F 0.15B)	Search + 2D
Unclustered Hash Index	BD(R+0.125)	2D	D(0.125B + # matches)	4D	4D

Heap file with unclustered hash index

Cost of scanning data entries is 1.25(0.1B)·D

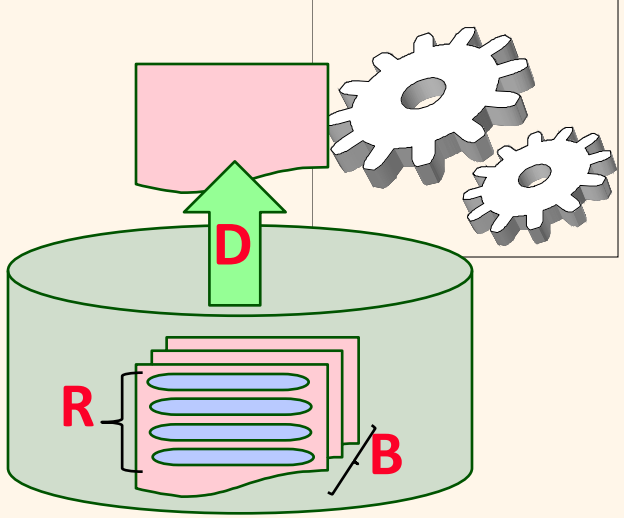
Hash structure cannot help

2D to update the index file + 2D to update the data file

2D to update the index file + 2D to update the data file

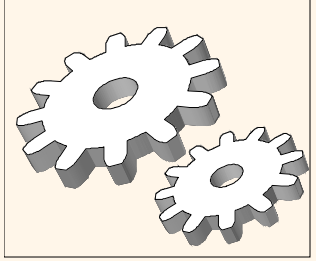
Cost of Operations

Several assumptions underlie these (rough) estimates!



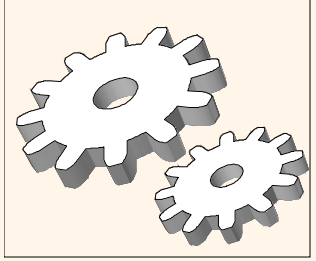
	Scan	Equality	Range	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search + D
Sorted	BD	$D \cdot \log_2 B$	$D(\log_2 B$ + # matching pages)	Search + BD	Search + BD
Clustered	1.5BD	$D \cdot \log_F 1.5B$	$D \cdot \log_F 1.5B$ + # matching pages	Search + D	Search + D
Unclustered Tree Index	BD(R+0.15)	$D(1 + \log_F 0.15B)$	$D \cdot (\log_F 0.15B$ + # matches)	$D(3 + \log_F 0.15B)$	Search + 2D
Unclustered Hash Index	BD(R+0.125)	2D	D(0.125B + # matches)	4D	4D

Understanding the Workload



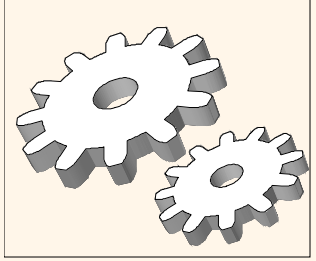
- For each **query** in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each **update** in the workload:
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Choice of Indexes



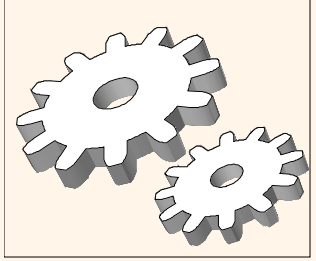
- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree?

Choice of Indexes (Contd.)



- One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!
  - For now, we discuss simple 1-table queries.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

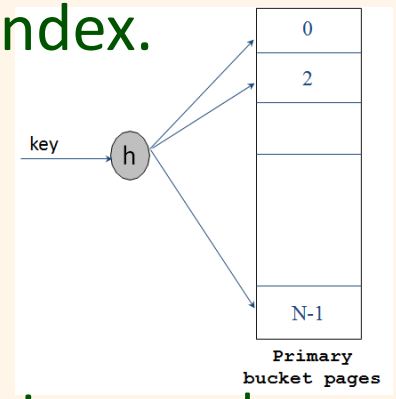
Index Selection Guidelines (1)



Attributes in WHERE clause are candidates for index keys.

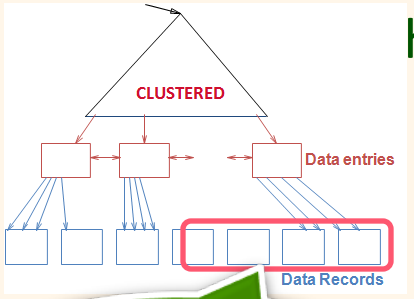
- Exact match condition suggests hash index.

```
SELECT   E.dno
FROM     Employees E
WHERE    E.num = 568429543
```



- Range query suggests tree index.

  Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.



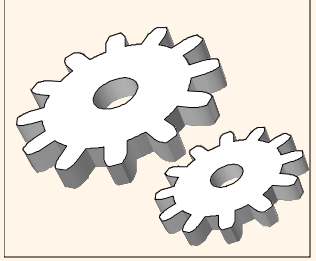
Employees older than 40

```
SELECT   E.dno
FROM     Employees E
WHERE    E.age > 40
```

```
SELECT   E.nam
FROM     Empl
WHERE    E.dno=
```

Dept. 123 has many employees

Index Selection Guidelines (2)

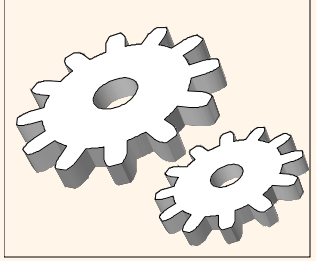


Multi-attribute search keys should be considered when a WHERE clause contains several conditions

```
SELECT   E.name
FROM     Employees E
WHERE    E.age = 20  AND  E.sal = 50000
                  1              2
```

- An index on <*age,sal*> would be better than an index on *age* or an index on *sal*
- Reason: The qualifying data entries are grouped together

Index Selection Guidelines (3)

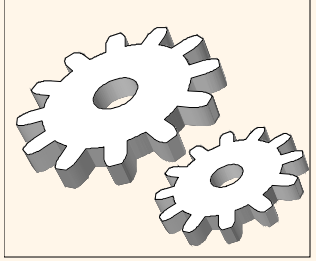


- Indexes can sometimes enable index-only evaluation for important queries.

  Example: use leaf pages of index on *age* to compute the average age

- Try to choose indexes that benefit as many queries as possible.
- Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Examples of Clustered Indexes (1)

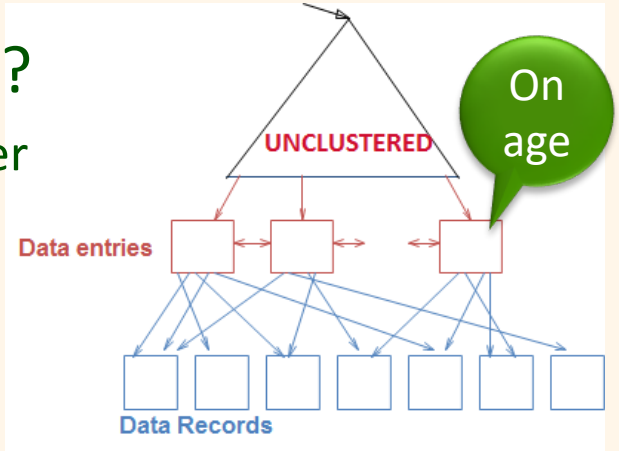


B+ tree index on E.age can be used to get qualifying tuples

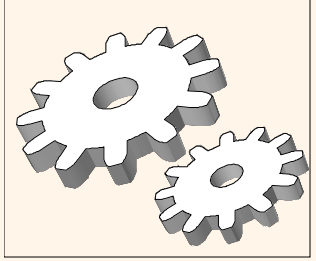
```
SELECT  E.dno
FROM    Emp E
WHERE   E.age>30
```

What is the selectivity of the condition ?

- If **most employees are older than 30**, a sequential scan of the relation would do almost as well
  - What if **only 10% are older than 30** ?
    - **Unclustered index**: Performing one I/O per qualifying tuple could be more expensive than a sequential scan
    - **Clustered index**: This is a good option



Examples of Clustered Indexes (2)

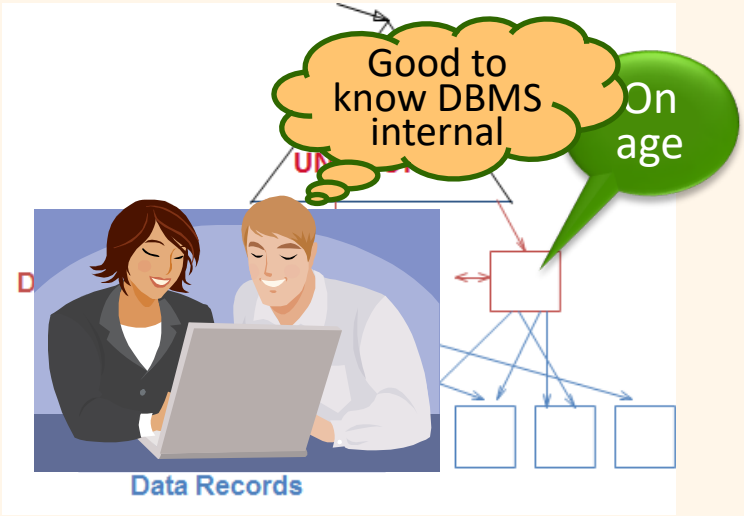


Using *E.age* index may be costly

➢Retrieve tuples with E.age > 25

➢Sort the tuples on dno, and count number of tuples for each dno

```
SELECT  E.dno,  COUNT (*)
FROM    Emp E
WHERE   E.age>25
GROUP BY E.dno
```

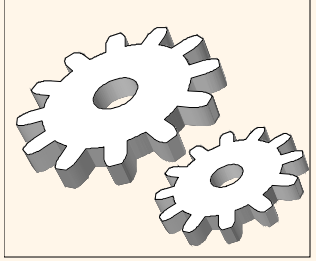


Clustered *E.dno* index may be better !

➢For each dno, count the tuples with *E.age* > 25

➢No need to sort !

Examples of Clustered Indexes (3)



```
SELECT  E.dno
FROM    Emp E
WHERE   E.hobby=Stamps
```

1

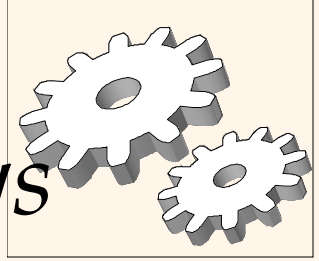
- If many employees collect stamps, a clustered index on *E.hobby* is helpful
  - If this query is important, we should consider this index

- An unclustered index on *E.eid* is good enough for the second query since no two employees have the same *E.eid*.

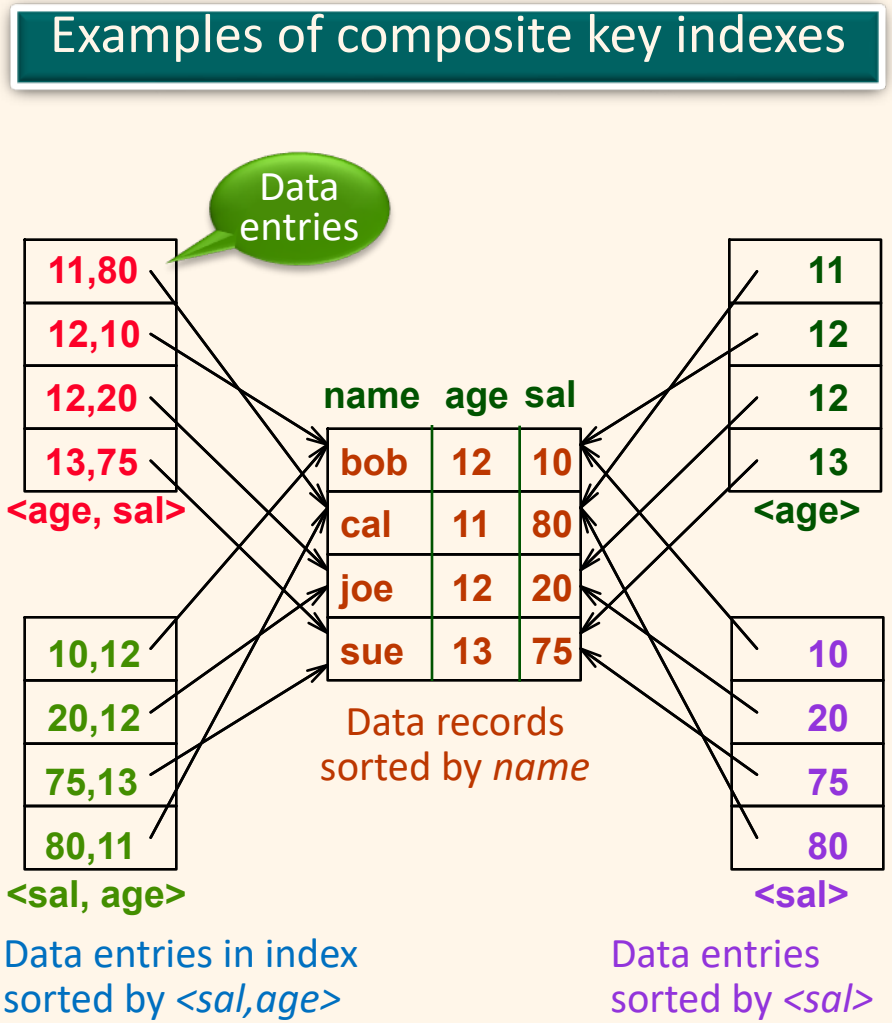
```
SELECT  E.dno
FROM    Emp E
WHERE   E.eid=328169455
```

2

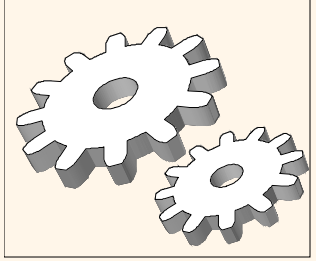
Indexes with Composite Search Keys



- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value, e.g. wrt <sal,age> index: age=20 and sal =75
  - Range query: Some field value is not a constant, e.g., age =20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.



Composite Search Keys

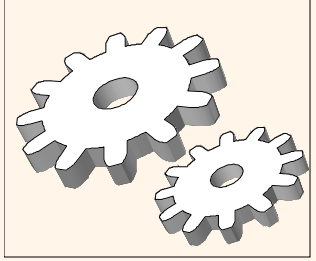


- Order of attributes is important for range queries

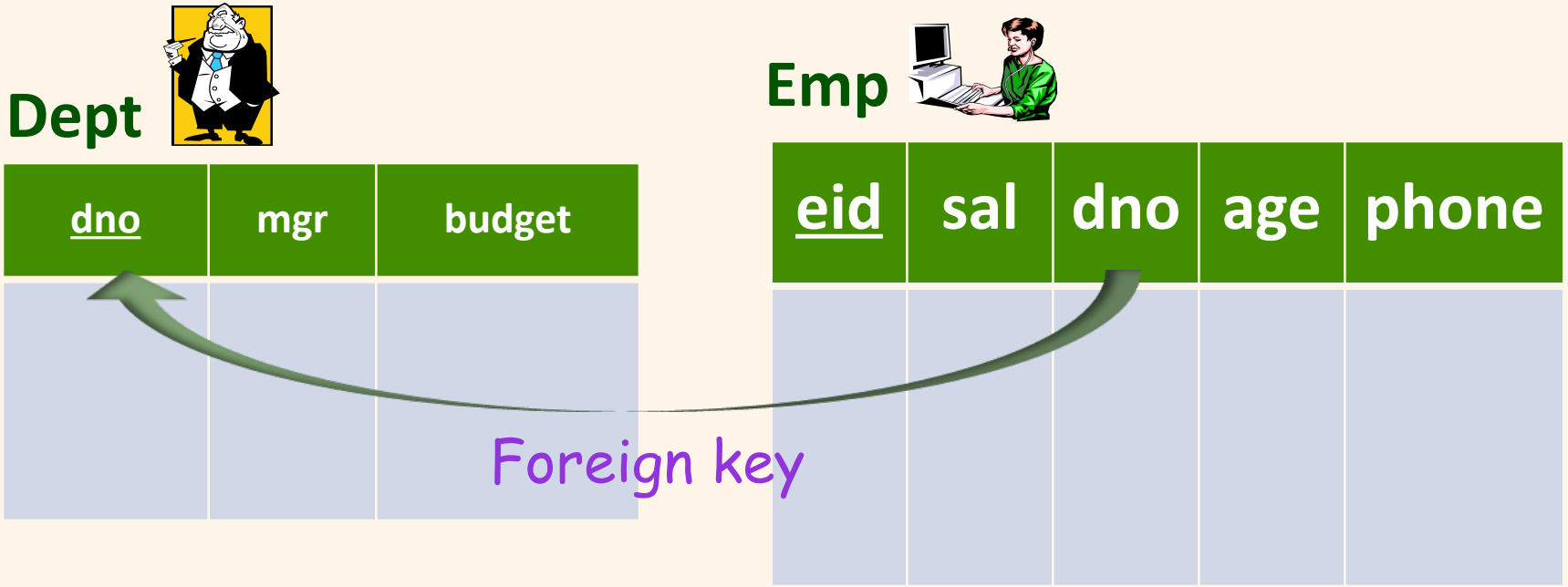
  key<*age*,*sal*> ≠ key<*sal*,*age*>

- To retrieve Emp records:
  - If condition is: *age*=30 AND 3000<*sal*<5000:
    - ➢Clustered <*age,sal*> index much better than <*sal,age*> index!
    - ➢Reason: The qualifying data entries are grouped together in <*age,sal*> index, but not in <*sal,age*> index
  - If condition is: 20<*age*<30 AND 3000<*sal*<5000:
    - ➢Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- Composite indexes are larger, updated more often

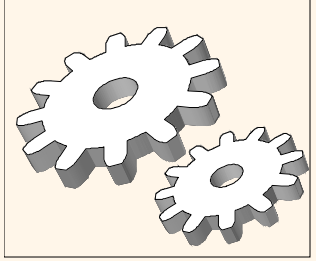
Database Example



We use this database for the following query examples



Index-Only Plans (1)



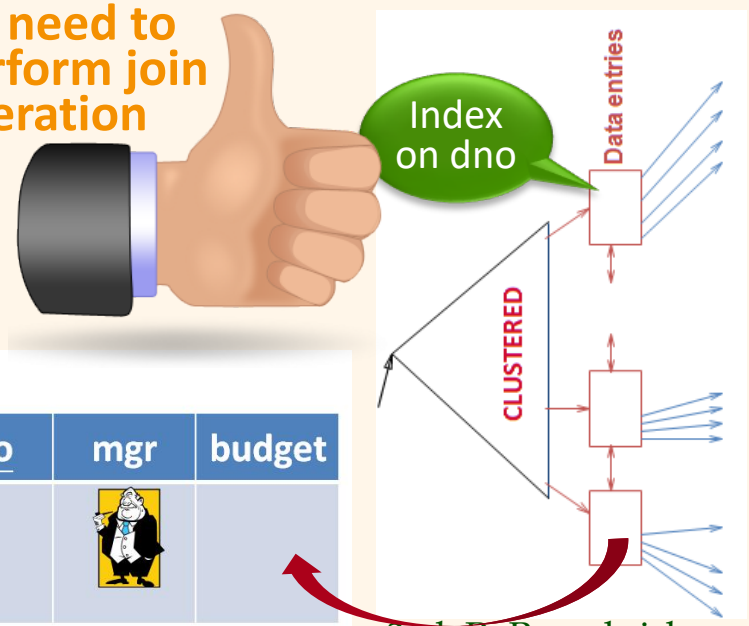
A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

```
SELECT  D.mgr
FROM    Dept D, Emp E
WHERE   D.dno=E.dno
```

Find mangers of departments with at least one employee

If <*E.dno*> index is available, no need to retrieve Employees tuples, i.e., scan the *E.dno* data entries and pick up the corresponding *Dept* tuples

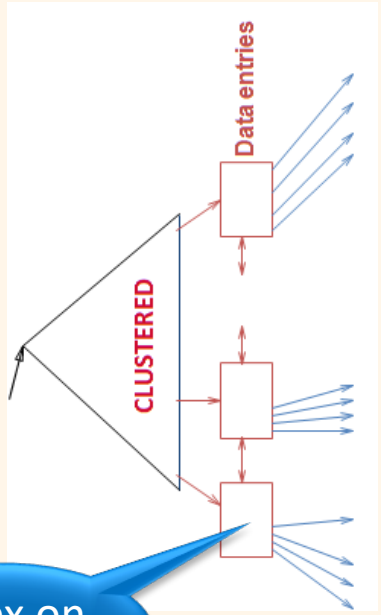
No need to perform join operation



Emp

eid	sal	dno	age	phone
				

Index-Only Plans (2)



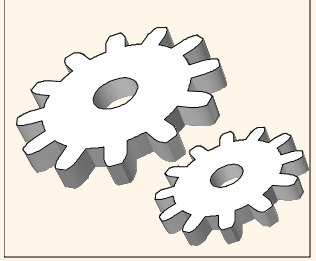
Emp

eid	sal	dno	age	phone

```
SELECT    E.dno, COUNT(*)
FROM      Emp E
GROUP BY  E.dno
```

If <*E.dno*> index is available, need only scan the data entries and count employees for each dno

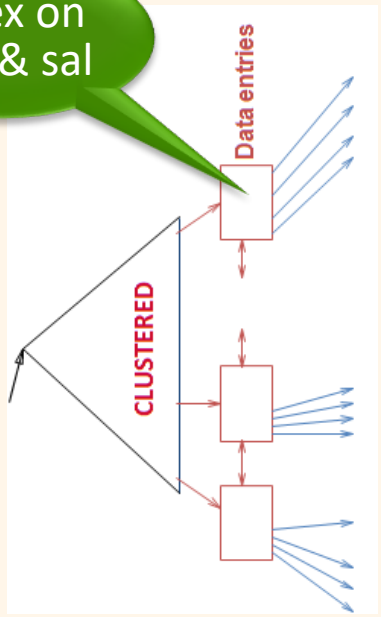
Index-Only Plans (3)



```
SELECT      E.dno, MIN(E.sal)
FROM        Emp E
GROUP BY    E.dno
```

If <*E.dno*,*E.sal*> tree index is available, need only scan the data entries and compute MIN(*E.sal*) for each *dno*

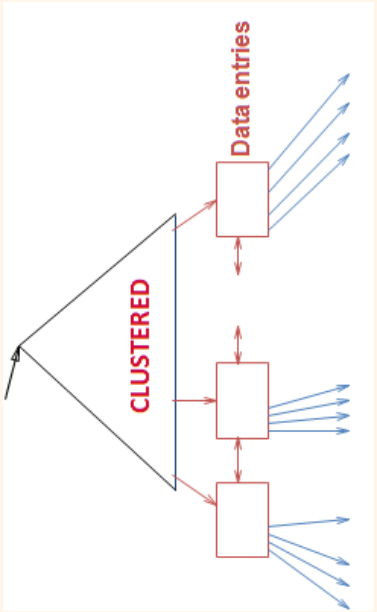
Index on dno & sal



Emp

eid	sal	dno	age	phone

Index-Only Plans (4)



Emp

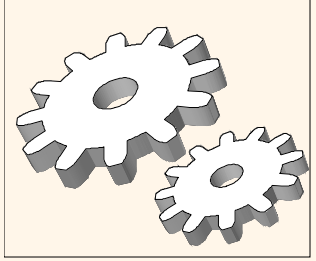
eid	sal	dno	age	phone

If <*E.age*,*E.sal*> or <*E.sal*,*E.age*> tree index is available, the average salary can be computed using only data entries in the index

Compute average salary of young executives

```
SELECT   AVG(E.sal)
FROM     Emp E
WHERE    E.age=25 AND
  E.sal BETWEEN 300000 AND 500000
```

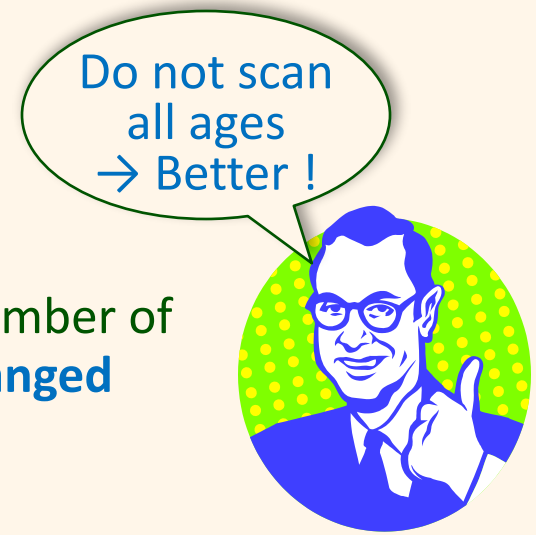
Index-Only Plans (5)



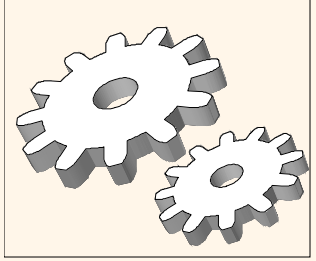
Index-only plans are possible if the key is <*dno*,*age*> or we have a tree index with key <*age*,*dno*>

```
SELECT      E.dno,  COUNT (*)
FROM        Emp E
WHERE       E.age=30
GROUP BY    E.dno
```

- **Using <*dno*,*age*> index**
  - Scan <u>all</u> data entries
  - For each *dno*, count number of tuples with *age*=30
- **Using <*age*,*dno*> index**
  - <u>Use index</u> find first data entry /w *age* = 30
  - Scan data entries with *age* = 30, and count number of tuples for each *dno* **(the departments are arranged continuously for age=30)**



Index-Only Plans (6)



Index only plans are possible

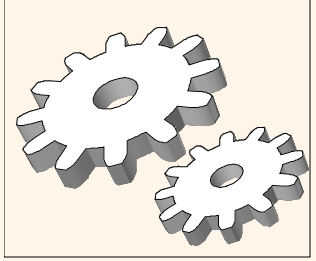
- **Using the <*age*,*dno*> index**
  - For data entries with *age* > 30, sort them on *dno*
  - Scan sorted data entries and count number of data entries for each *dno*
- **Using <*dno*,*age*> index**
  - Scan data entries
  - For each *dno,* count number of data entries with *age* > 30

```
SELECT    E.dno, COUNT (*)
FROM      Emp E
WHERE     E.age>30
GROUP BY  E.dno
```

Note: Sorting is not necessary if the department counters can fit in memory

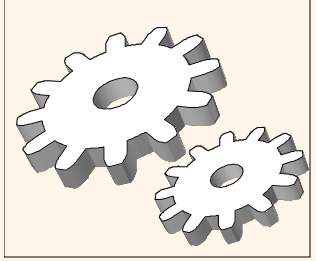
No sorting. Better !

Summary



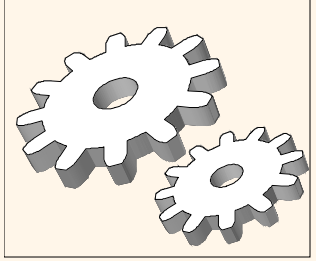
- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries (<key, rid> pairs or <key, rid-list> pairs) plus a way to quickly find entries (using hashing or a search tree) with given key values.

Summary (Contd.)



- Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.
  - Choice orthogonal to *indexing technique* used to locate data entries with a given key value.
- Can have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as clustered vs. unclustered, and primary vs. secondary. Differences have important consequences for utility/ performance.

Summary (Contd.)



- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates? What attributes/ relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.