

CptS 451- Introduction to Database Systems

Storage and Indexing (DMS Ch-9.5-9.7 and Ch-8)

Instructor: Sakire Arslan Ay



World Class. Face to Face.

Database Management Systems – Ramakrishnan and Gehrke



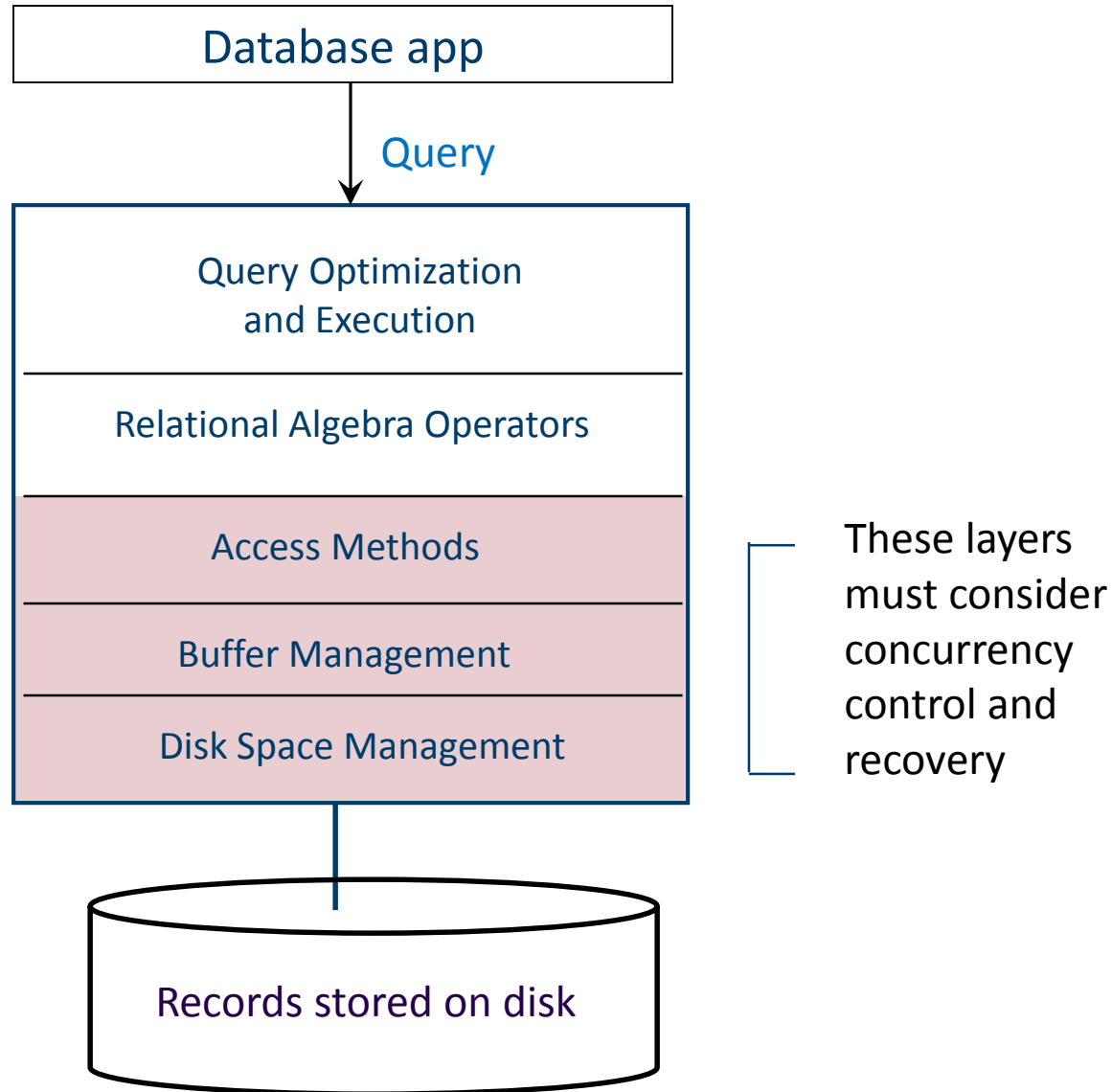
Storage and Indexing (topics)

- DBMS Architecture Overview (this video)
- File Management (this video)
- Storage and Indexing (next video)

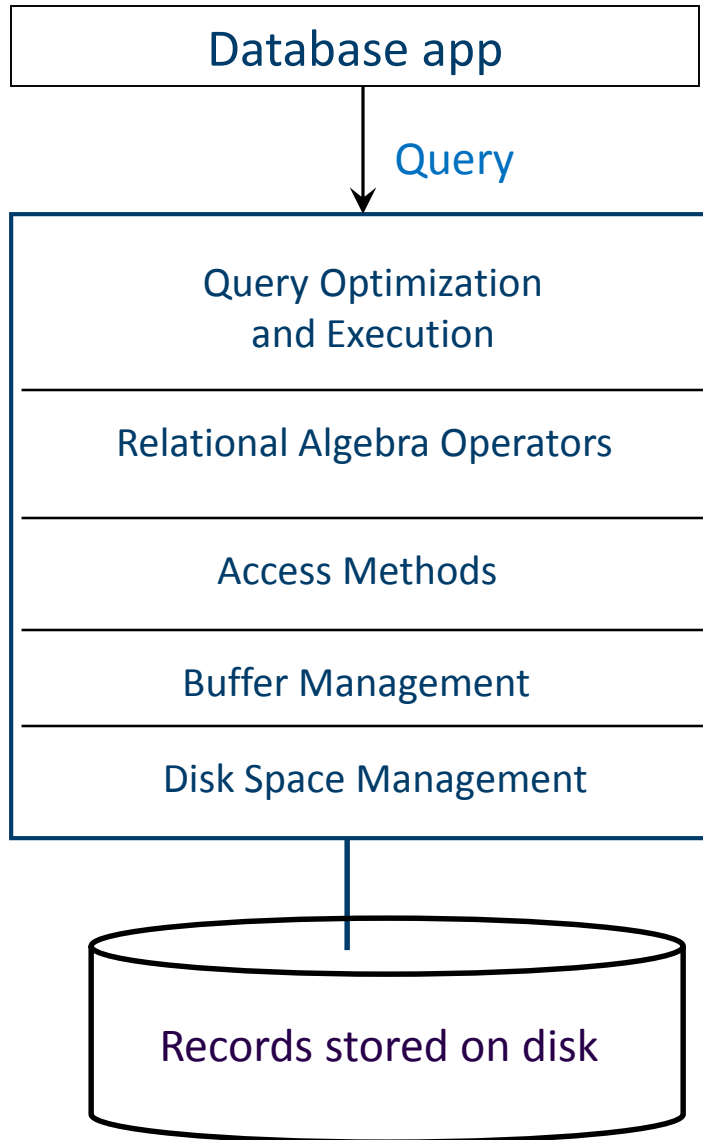
Later:

- Database tuning and choosing indexes

DBMS Architecture Overview



Where are we?



Lecture Notes-13: Query Execution

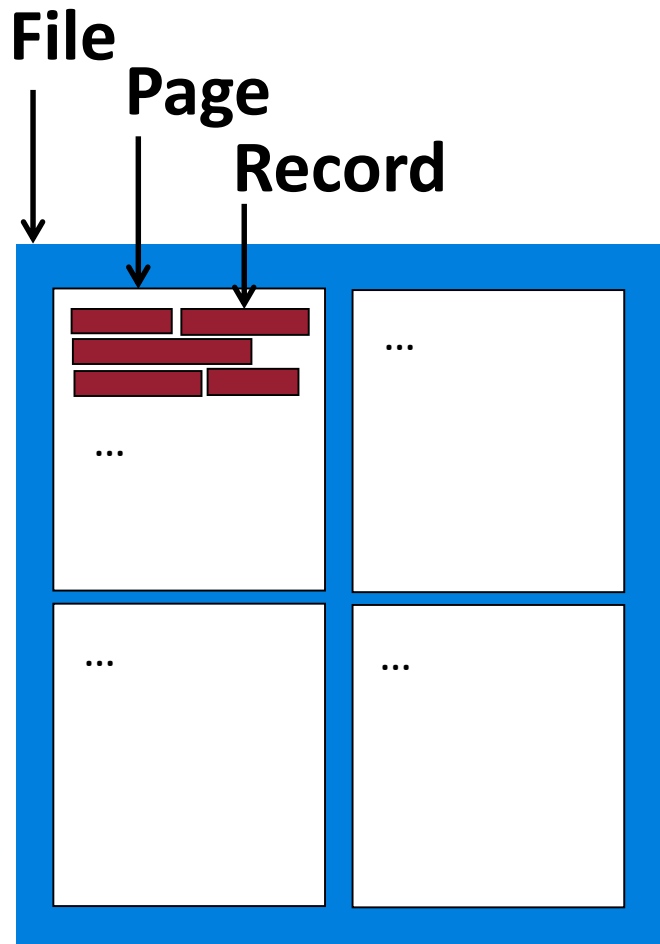
Lecture Notes-11:
Storage and Indexing

Lecture Notes-12:
Transaction Management and
Concurrency

File and Disk Management

Data Storage

- How is data of a relation stored on disk?



FILE: A collection of **pages**, each containing a number of records. The File API must support:

- insert/delete/modify** record
- fetch** a particular record (specified by *record id*)
- scan** all records (possibly with some conditions on the records to be retrieved)

PAGE: Fixed number of bytes

RECORD: Variable or fixed size (depends on schema)

Typically:

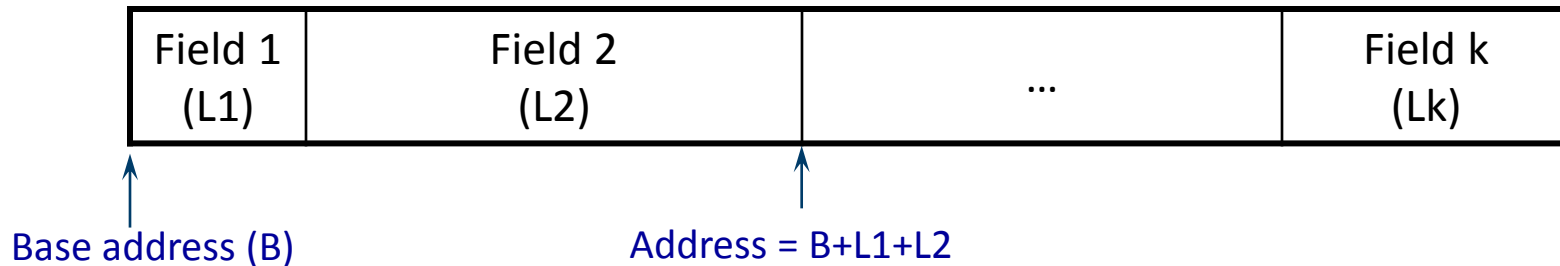
file page size = disk block size =
buffer frame size

Records

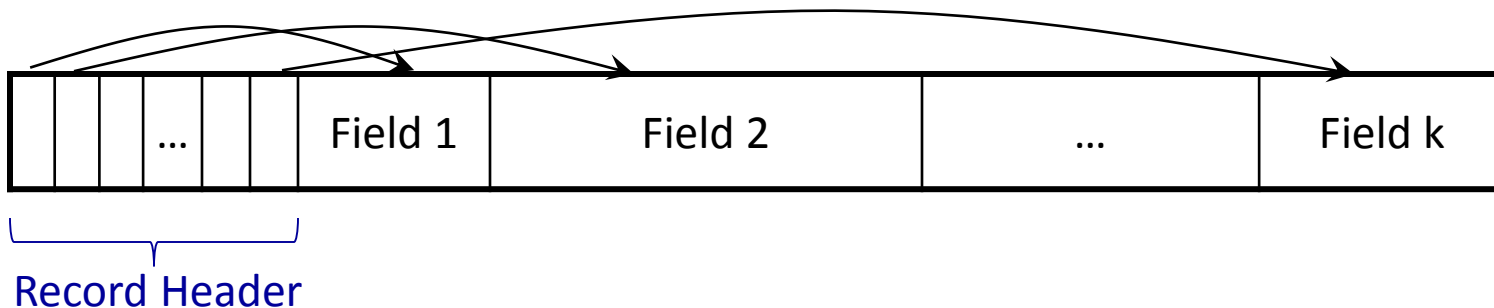
- Records represent tuples of a relation
- Records are stored in pages
- When a record need to be accessed or updated, the entire page (disk block) that the record is stored need to be moved into main memory.
- Record id = RID
 - Typically $RID = (pageID, slot\#)$
- Records:
 - Fixed length
 - Variable length

Record Formats

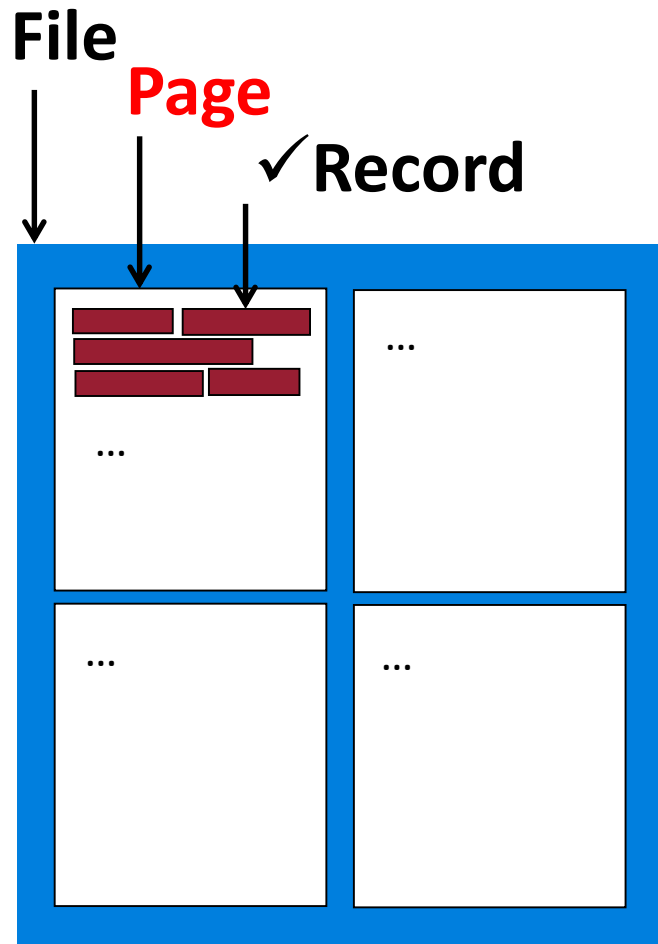
- Fixed-length records → Each field has the same length in all the records)



- Variable length records
 - Remarks: NULLS require no space at all; there is some directory overhead

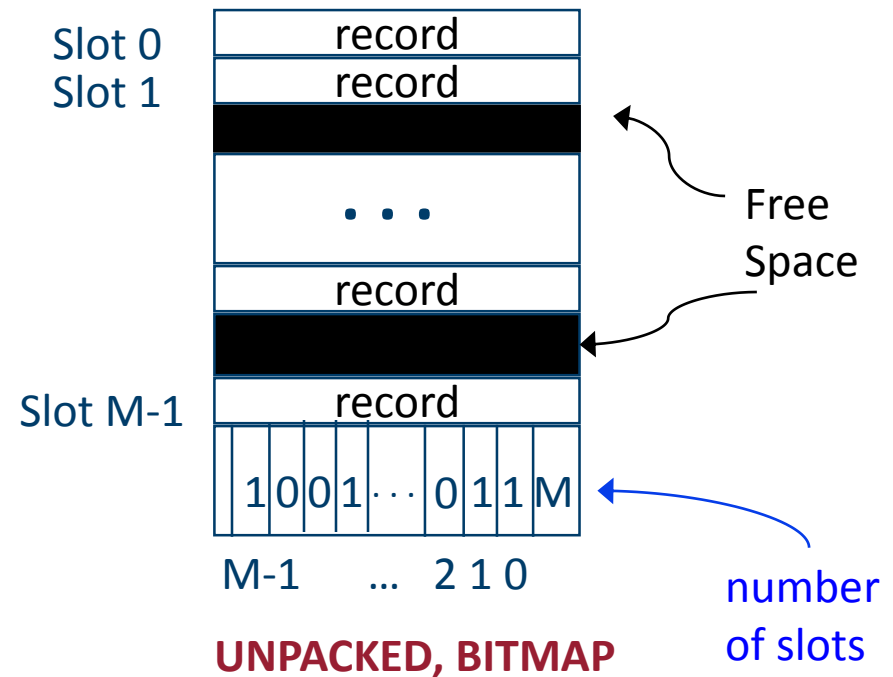
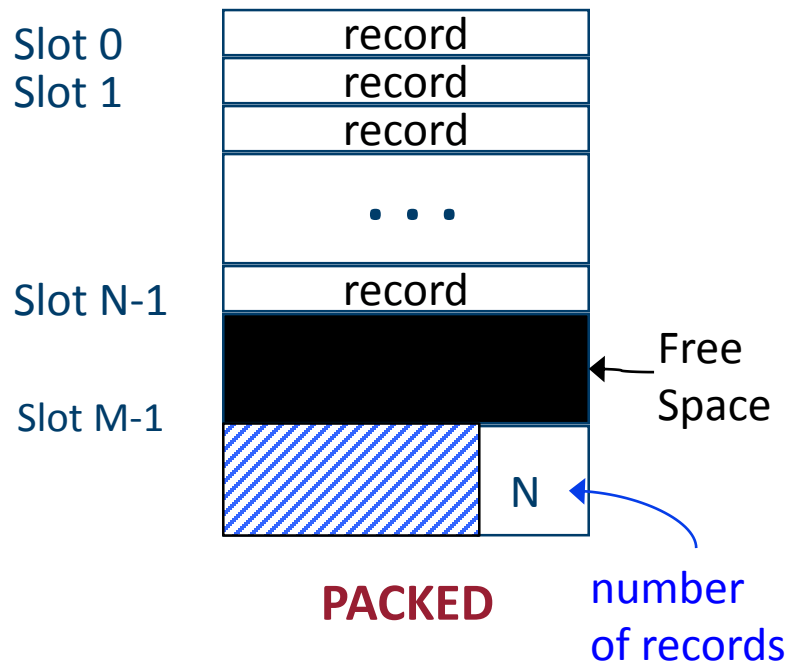


Data Storage



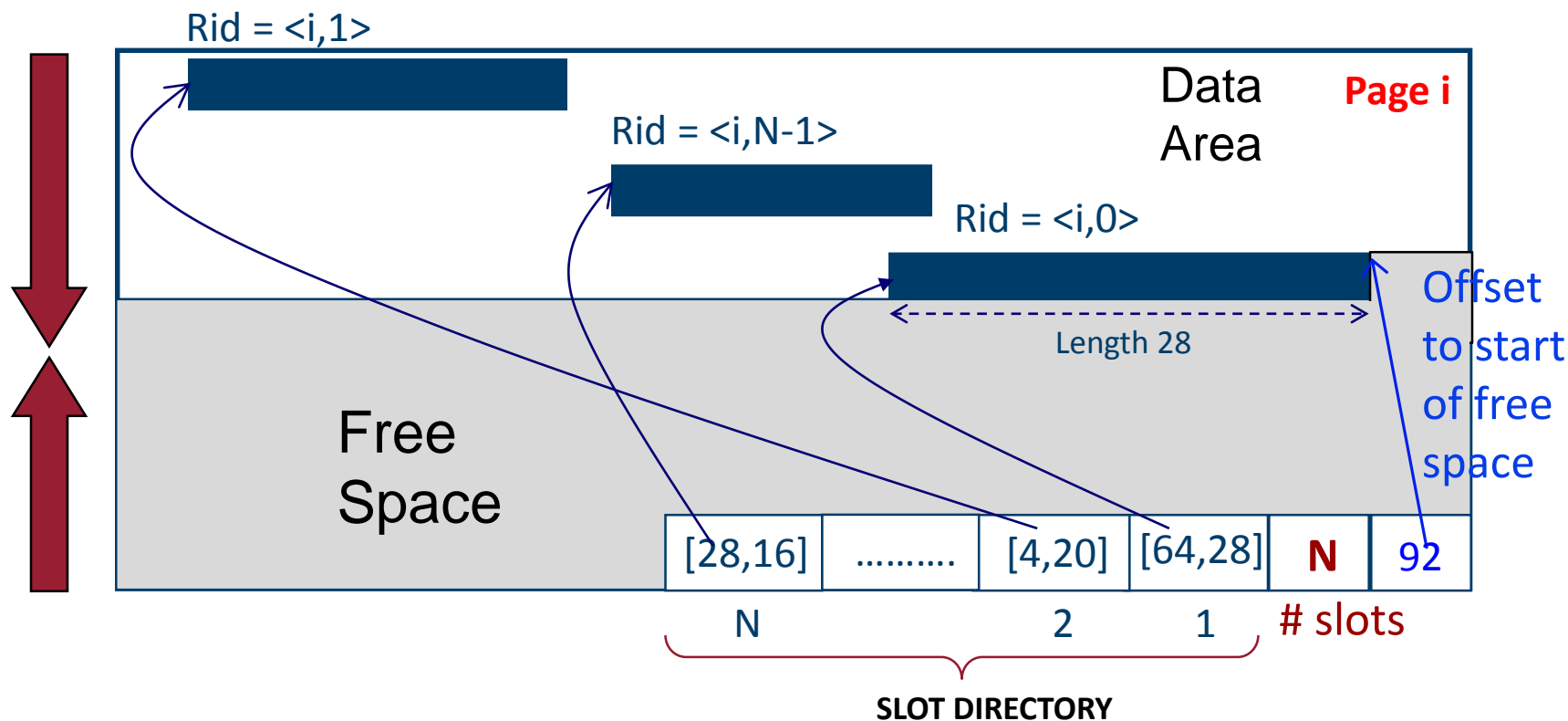
- **Next:** Page Formats

Page Formats: Fixed Length Records



- A page is a collections of slots, each containing a record.
- In first alternative (left), free space management requires record movement.
 - *Changes RIDs - may not be acceptable.*

Page Formats: Variable Length Records - "Slotted Page"

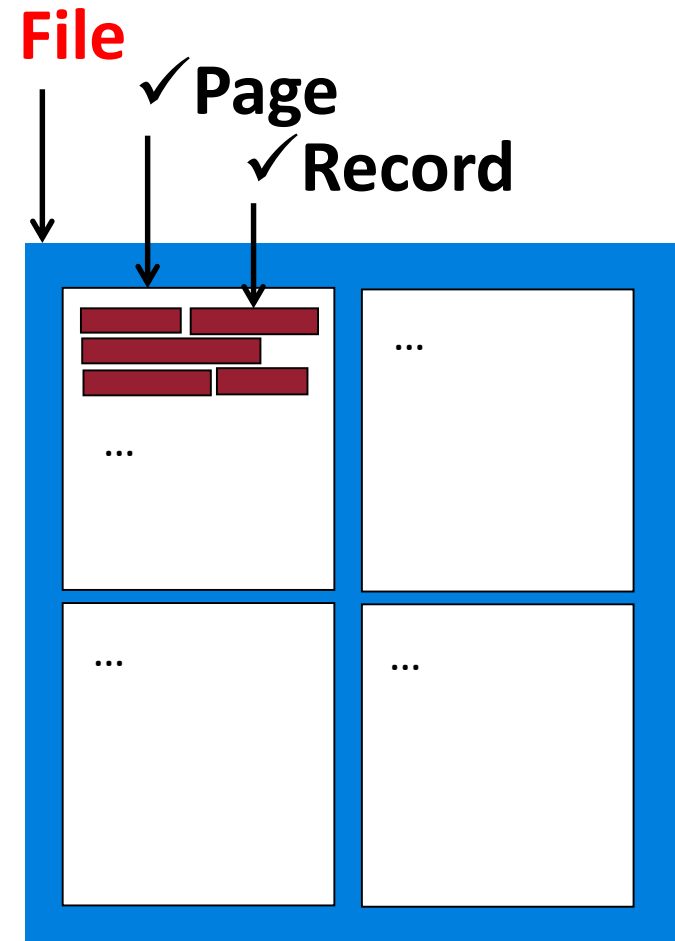


- Slot contains: *[offset (from start of page), length]*
 - both in bytes
- Record id = *<page id, slot #>*
- Page is full when data space and slot array meet.

So far we've organized:

- Fields into Records
(fixed and variable length)
- Records into Pages
(fixed and variable length)

Now we need to organize
Records and Pages into Files



Alternative File Organizations

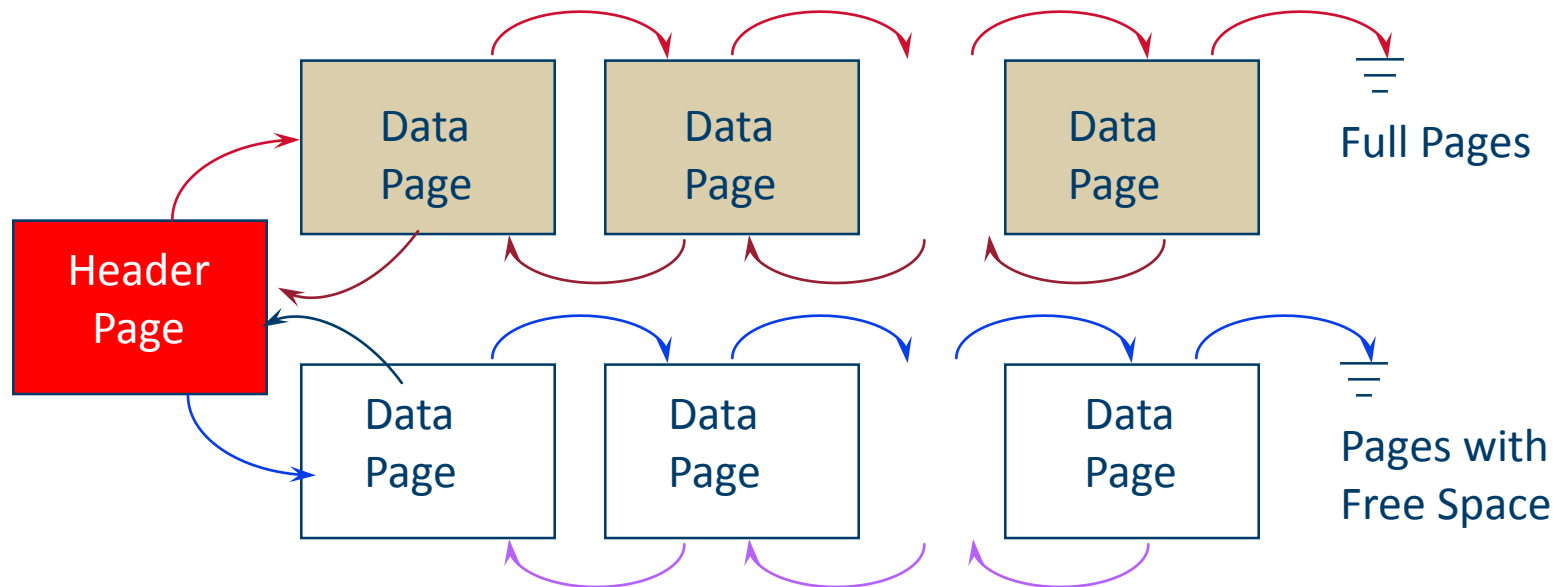
- Many alternatives exist, each good for some situations, and not so good in others:
 - **Heap files**: Unordered. Fine for file scan retrieving all records. Easy to maintain.
 - **Sorted Files**: Best for retrieval in *search key* order, or if only a 'range' of records is needed. Expensive to maintain.
 - **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.



Unordered (Heap) Files

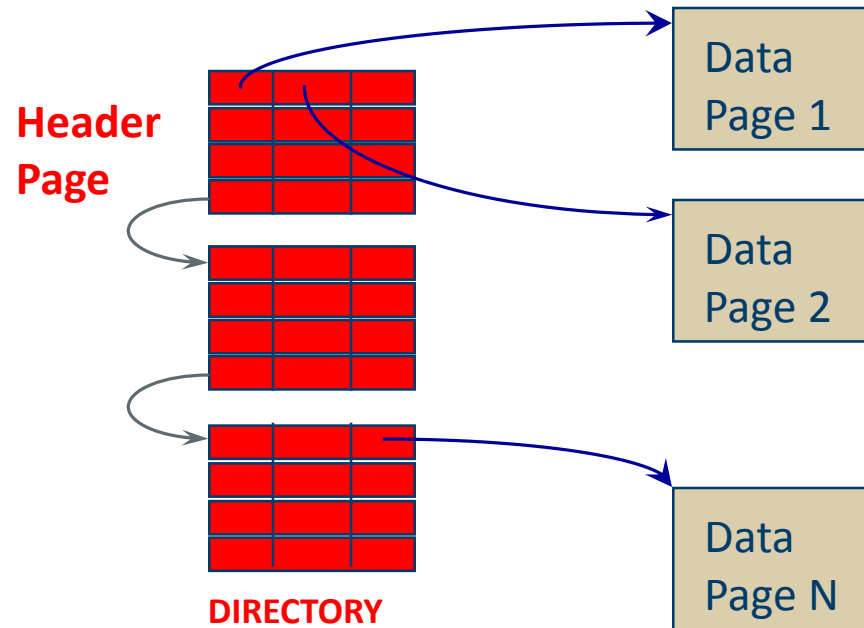
- Simplest file structure contains records **in no particular order**.
- As file grows and shrinks, pages are allocated and de-allocated.
- To support record level operations, we must:
 - keep track of the *pages* in a file
 - keep track of *free space* on *pages*
 - keep track of the *records* on a page
- Can organize pages as a list, as a directory, etc...

Heap File Implemented as a List



- The Heap file name and header page id must be stored persistently. (*<heap_file_name, header_page_addr>*)
 - The catalog is a good place for this.
- Each page contains 2 'pointers' plus data.
- Q: Disadvantage?

Heap File Using a Page Directory



- Directory is a collection of pages; linked list implementation is just one alternative.
- Directory may contain **free-space count** for each page.
 - Faster inserts for variable-length records
- **Q: How to find a particular record in a Heap file???**

Cost Model for Analysis

- Average-case analysis; based on several simplistic assumptions.
- We ignore CPU costs, for simplicity:
 - D:** The average time to read or write a disk page (initially we will ignore D)
 - B:** The number of pages (data blocks)
 - R:** Number of records per block
- Initially, we simply count number of disk block I/O's
 - ignores gains of pre-fetching and **sequential access**; thus, even I/O cost is only loosely approximated.

✉ ***Good enough to show some overall trends!***

Operations to Compare

- **Scan:** Fetch all records from disk
- **Equality search:** Fetch the pages that contain qualifying records from disk.
- **Range selection:** Fetch the pages that contain the records that satisfy a range selection.
- **Insert a record:** Identify the page where record should be inserted, fetch that page, modify it, write back to disk.
- **Delete a record**

Some Assumptions in the Analysis

- Single record insert and delete.
- Equality selection on key - exactly one match (what if more or less???)
- For Heap Files we'll assume:
 - Insert always appends to end of file.
 - Delete just leaves free space in the page.
 - Empty pages are not de-allocated.
 - If using directory implementation assume directory is in-memory.

Average Case I/O Counts for Operations (B = # of pages in file)



	Scan all records	Equality Search (avg) (1 match)	Range Search	Insert	Delete
Heap File					
Sorted File					



Searching in a Heap File

- File is not sorted on any attribute
- Consider relation:

`Student(sid: int, name: varchar(20), age: int, ...)`

1111	Jack	18	...
2222	Mary	21	...
1122	Joe	19	...
2221	John	20	...
3333	Mike	21	...
1113	Tom	21	...
3331	Jill	18	...
2223	Joy	19	...
1110	Sue	22	...
4440	Sam	18	...

→ 1 record

1 page

....

5550	Tom	19	...
4444	Tim	22	...
5555	Kate	20	...
6660	Kelly	17	...
7771	Cole	22	...
8881	Carl	18	...

- 10,000 students
- 10 student records per page
- Total number of pages: 1000 pages
- Find student whose **sid** is 5555
 - Must read on average 500 pages
- Find all students who are between 17 and 20 years old
 - Must read all 1000 pages

Average Case I/O Counts for Operations (B = # disk blocks in file)

	Scan all records	Equality Search (avg) (1 match)	Range Search	Insert	Delete
Heap File	B	$0.5B$	B	2	$0.5B+1$
Sorted File					

Sorted Files

- Heap files are **lazy** on **update** - you end up paying on searches.
- Sorted files **eagerly** maintain the file on **update**.
 - The opposite choice in the trade-off
- Assumptions for sorted file analysis:
 - No gaps allowed, pages fully packed always. Files compacted after deletions.
 - Searches are on sort key field(s).

Average Case I/O Counts for Operations (B = # disk blocks in file)

	Scan all records	Equality Search (1 match)	Range Search	Insert	Delete
Heap File	B	$0.5B$	B	2	$0.5B+1$
Sorted File					
Indexes					



Searching in a Sorted File

- File is sorted on an attribute, usually on the primary key (ex: sid)

Student(sid: int, name: varchar(20), age: int, ...)

1110	Sue	22	...
1111	Jack	18	...
1113	Tom	21	...
1122	Joe	19	...
2221	John	20	...
2222	Mary	21	...
2223	Joy	19	...
3331	Jill	18	...
3333	Mike	21	...
4440	Sam	18	...

....

5000	Tim	22	...
5550	Tom	19	...
5555	Kate	20	...
6660	Kelly	17	...
7771	Cole	22	...
8881	Carl	18	...

- Total number of blocks: 1,000 blocks
- Find student whose **sid** is 5555
 - Could do binary search, read $\log_2(1,000) \approx 10$ blocks
- Find all students with $sid > 5000$
- Find all students older than 20
- Note: Sorted files are inefficient for inserts/deletes

Average Case I/O Counts for Operations (B = # disk blocks in file)

	Scan all records	Equality Search (1 match)	Range Search	Insert	Delete
Heap File	B	0.5B	B	2	0.5B+1
Sorted File	B	$\log_2 B$ (if on sort key) $0.5B$ (otherwise)	$\log_2 B +$ (selectivity * B)	$\log_2 B + B$	$\log_2 B + B$
Indexes					

Average Case I/O Counts for Operations

(B = # disk blocks in file, D=avg time to read/write a disk page)



	Scan all records	Equality Search (1 match)	Range Search	Insert	Delete
Heap File	BD	0.5BD	BD	2D	0.5D(B+1)
Sorted File	BD	$(\log_2 B)D$ (if on sort key) $0.5BD$ (otherwise)	$D(\log_2 B + (\text{selectivity} * B))$	$D(\log_2 B + B)$	$D(\log_2 B + B)$
Indexes					

File Structure Summary

- File Access layer manages access to records in pages.
 - Record and page formats depend on fixed vs. variable-length.
 - Free space management is an important issue.
- Many alternative file organizations exist, each appropriate in some situation.
 - We looked at Heap and Sorted so far.
 - If selection queries are frequent, sorting the file or building an *index* is important.
- Our cost calculations are imprecise, but can expose fundamental systems tradeoffs.
- **Next up: Indexes.**

Indexing: Motivation

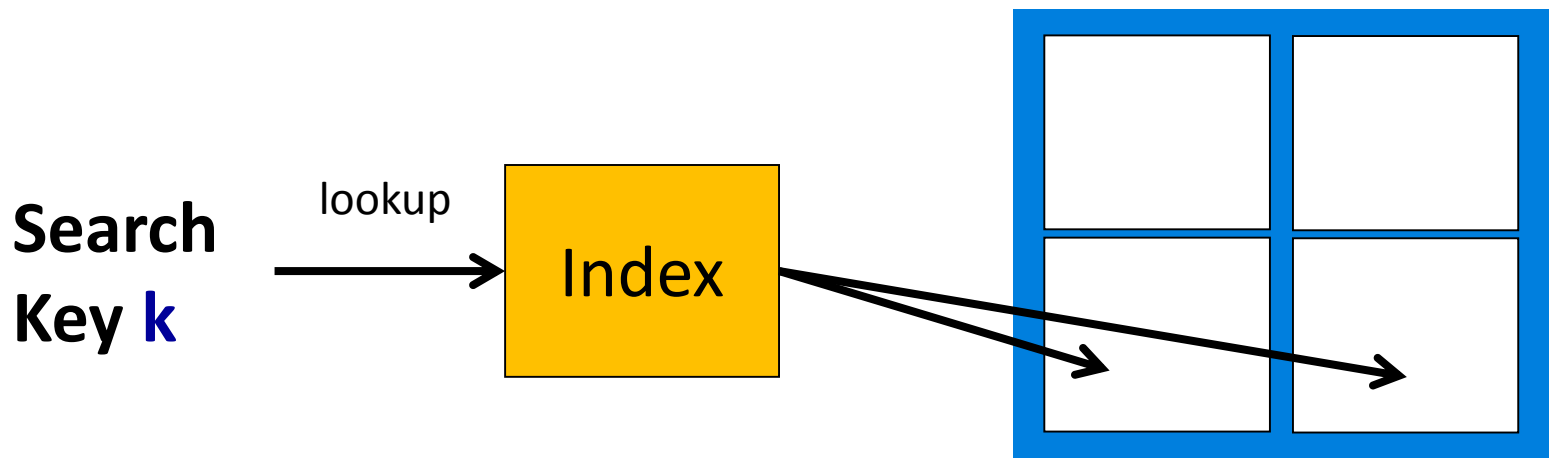
How to answer queries on relation R?

```
SELECT * FROM R WHERE R.name = "something";  
SELECT * FROM R WHERE R.B >= 5 AND R.B <= 10;
```

1. Scan file, reading blocks one by one
 2. For each tuple in block check condition
- Inefficient if very few tuples satisfy condition!
 - The relation may be huge...

Indexes

- 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 the *key* of the relation.
(However the indexes are often built on the relation *keys* – why?)

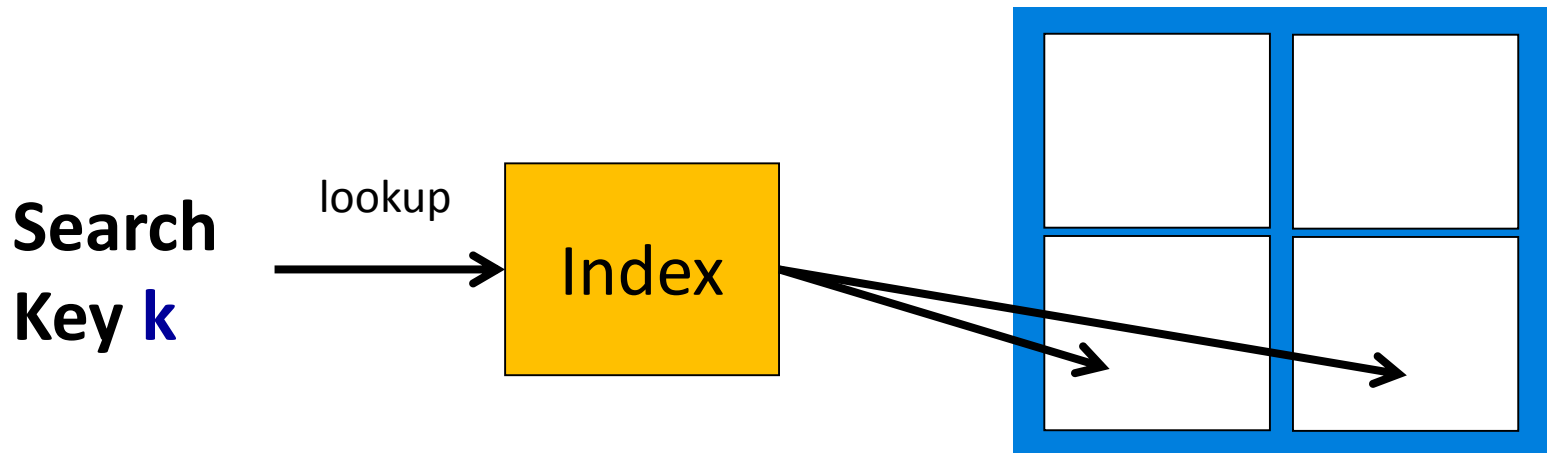


Indexes

- An index contains a collection of *data entries* (k^*) – usually– of the form:

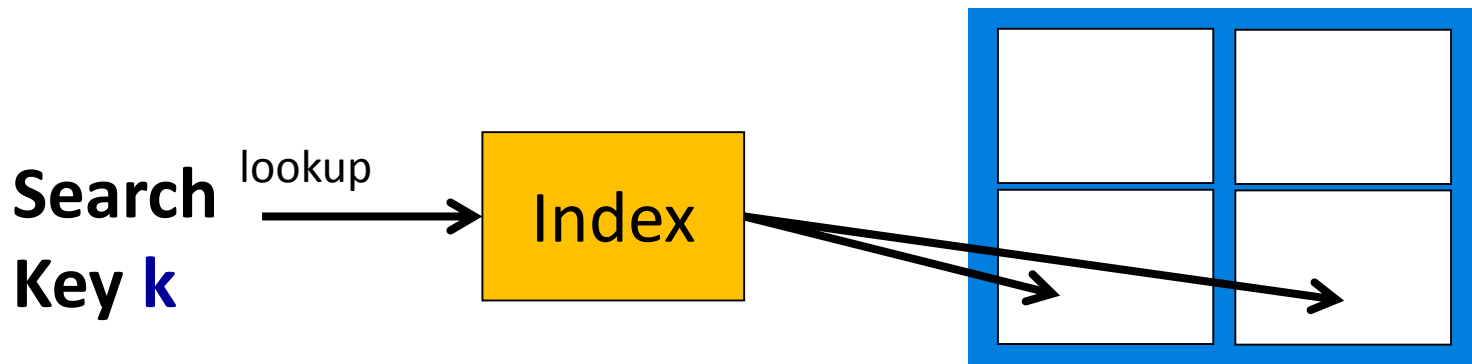
<i>search key</i>	pointer to rid(s)
-------------------	-------------------

- It supports efficient retrieval of all data entries k^* with a given key value k .
- Given data entry k^* in the index, we can find the actual record (with key k) in at most one disk I/O.



Alternatives for Data Entry k^* in Index

- In a data entry k^* we can store:
 - Data record with key value k , or *(alternative 1)*
 - $\langle k, rid \text{ of data record with search key value } k \rangle$, or *(alternative 2)*
 - $\langle k, \text{list of rids of data records with search key } k \rangle$ *(alternative 3)*
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value k .



Alternatives for Data Entries (Contd.)

- **Alternative 1:**

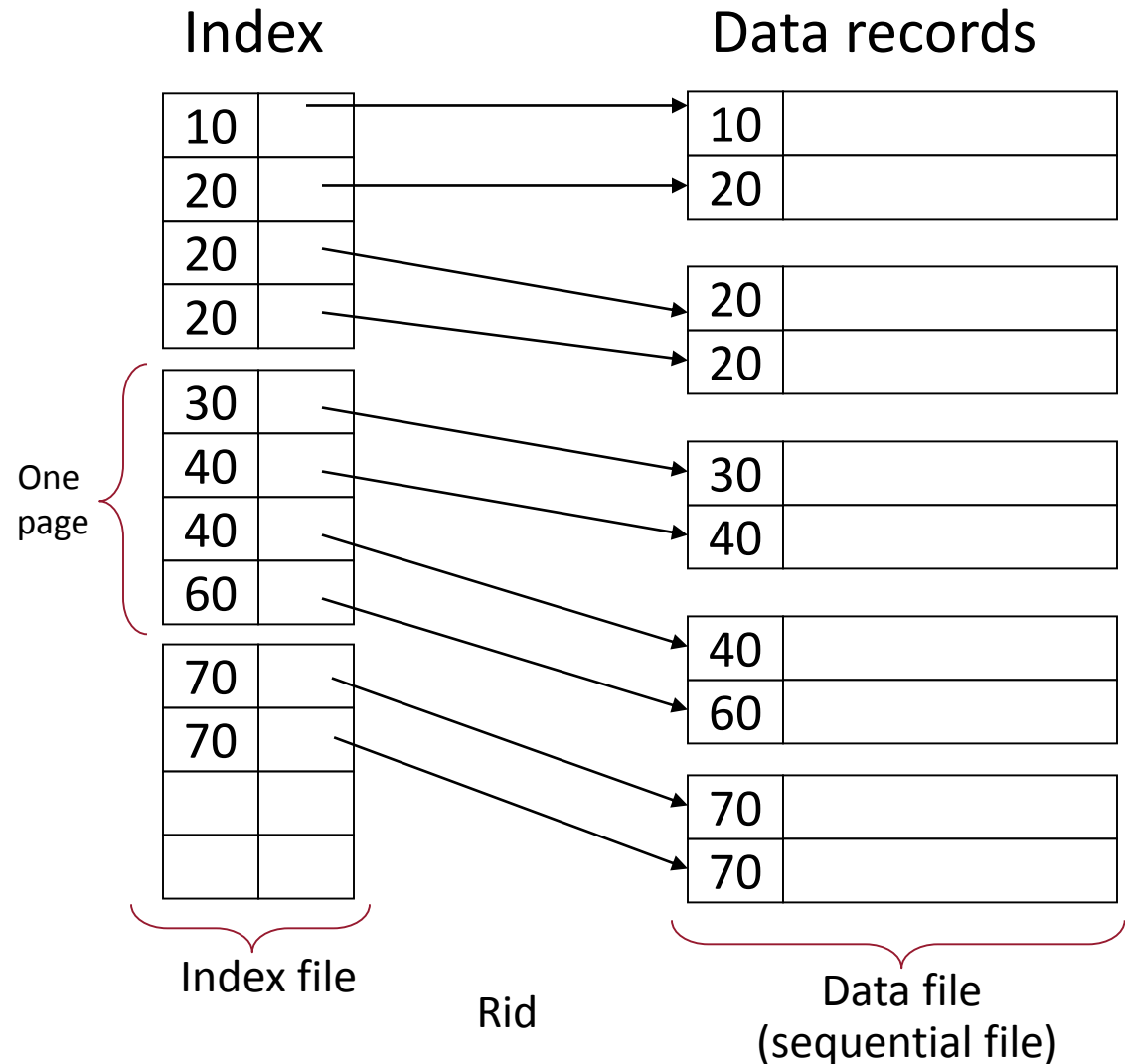
- If this is used, index structure is a file organization for data records.
- At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated)

- **Alternatives 2 and 3:**

- Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small.
- Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Alternative2- Example

- Each entry in index is a pair of :
(Search Key, **Rid** Pointer)
- Every **record in the data file** has an entry in the index
- Index file is **much** smaller than the data file, since we only store the key Rid, not the record content

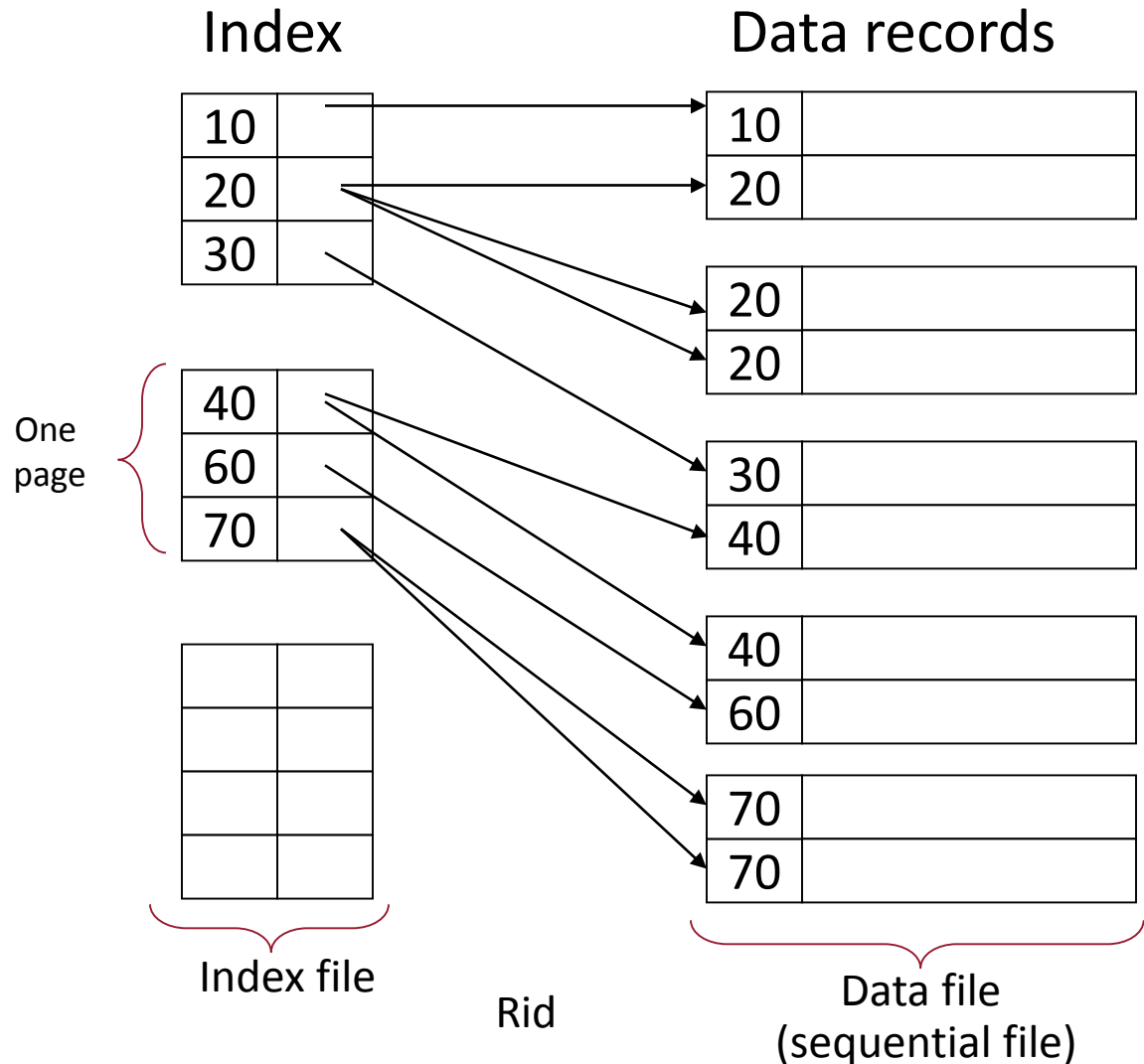


Alternative3- Example

- Each entry in index is a pair of :
(Search Key, List of **Rid** Pointers)

- Every **search key value** has an entry in the index

- Index file is **much** smaller than the data file, since we only store the key Rid, not the record content



Index Classification



1. Primary vs. secondary:

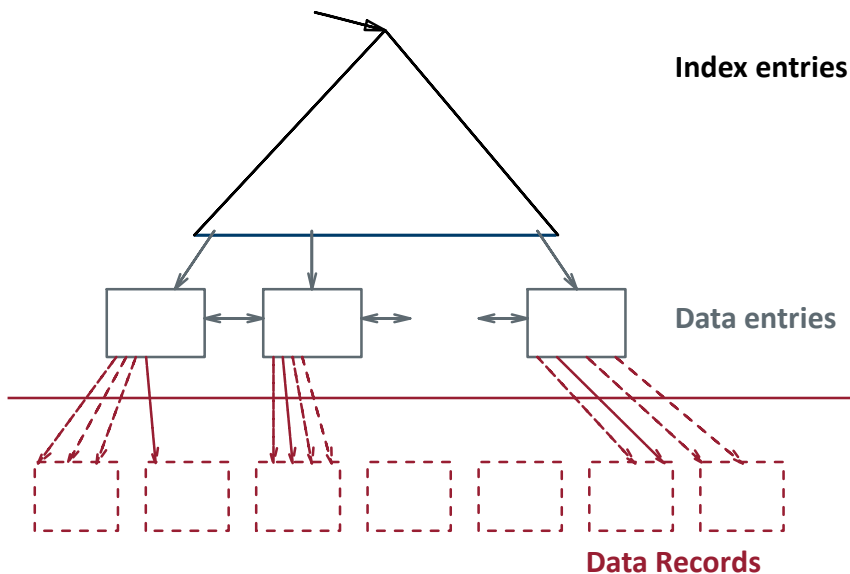
- If search key is the primary key, then called primary index. The index is guaranteed not to have duplicate search key values
- Otherwise called secondary key. May have duplicates.
- **Unique index**: A secondary index with no duplicate search keys. (i.e., search key contains a candidate key).

2. Clustered vs. unclustered:

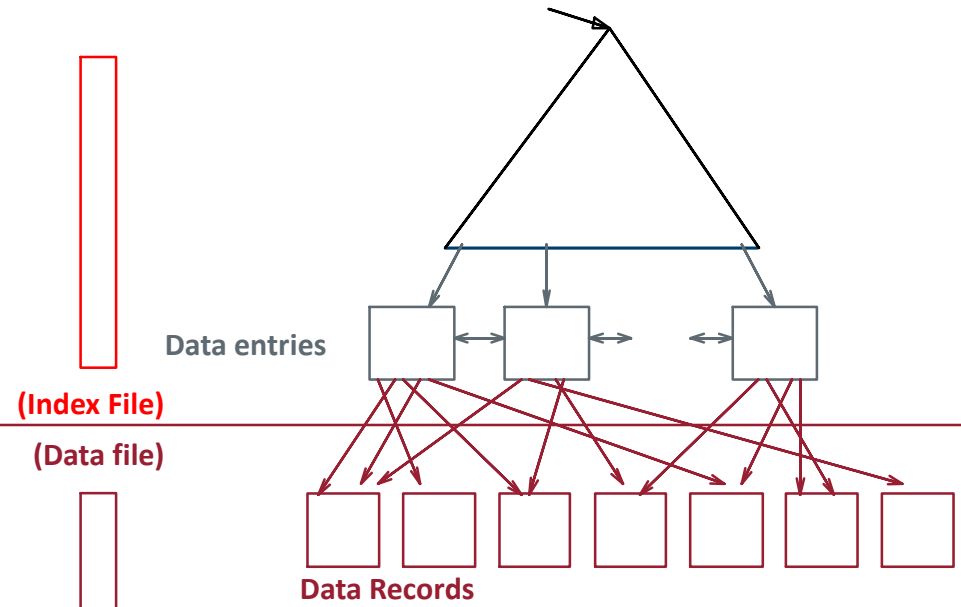
- If order of data records is the same as, or 'close to', order of data entries, then called clustered index.
 - A file can be clustered on at most one search key.
 - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).
 - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
 - How about cost of inserting data records?

Clustered vs. Unclustered Index

CLUSTERED



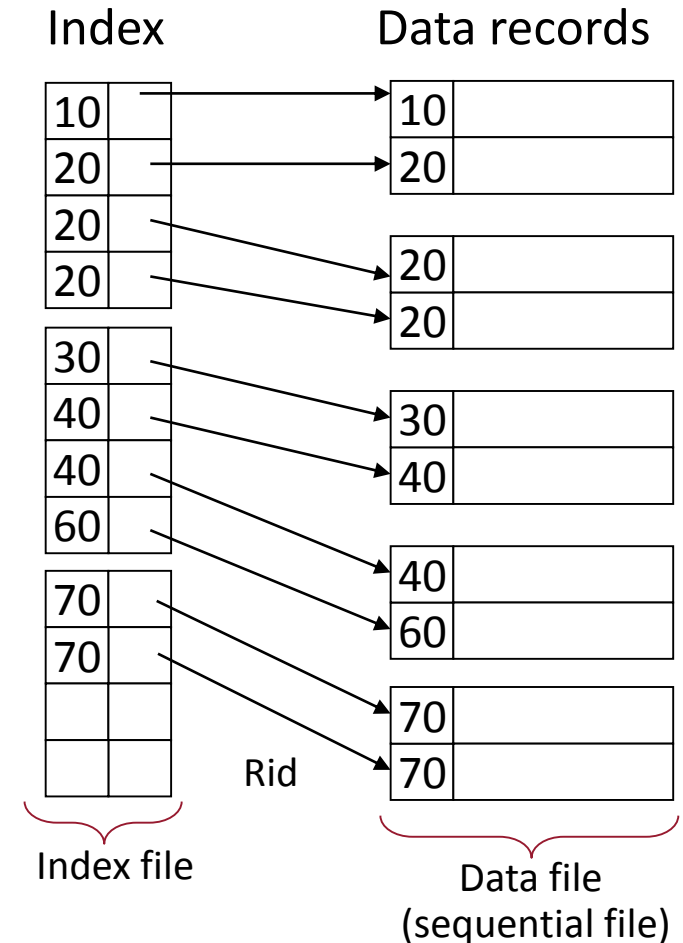
UNCLUSTERED



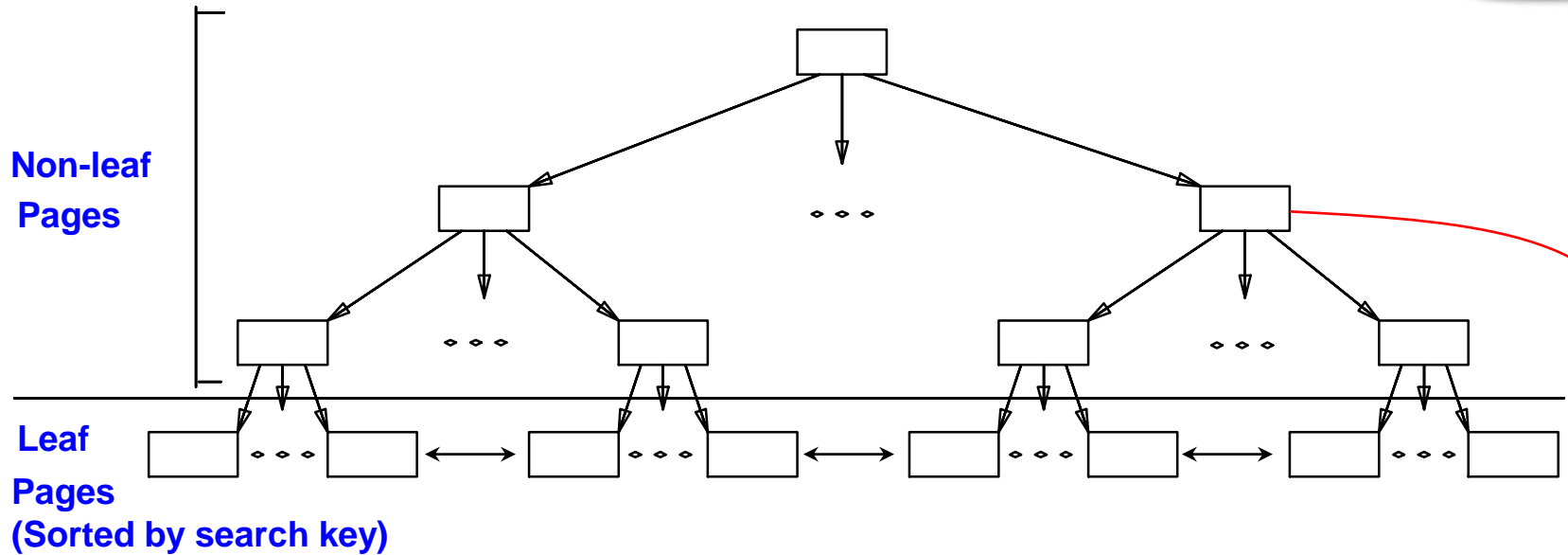
- Clustered = records close in index are close in data
- Generally, in a clustered B+ Tree index data entries are data records

Large Indexes

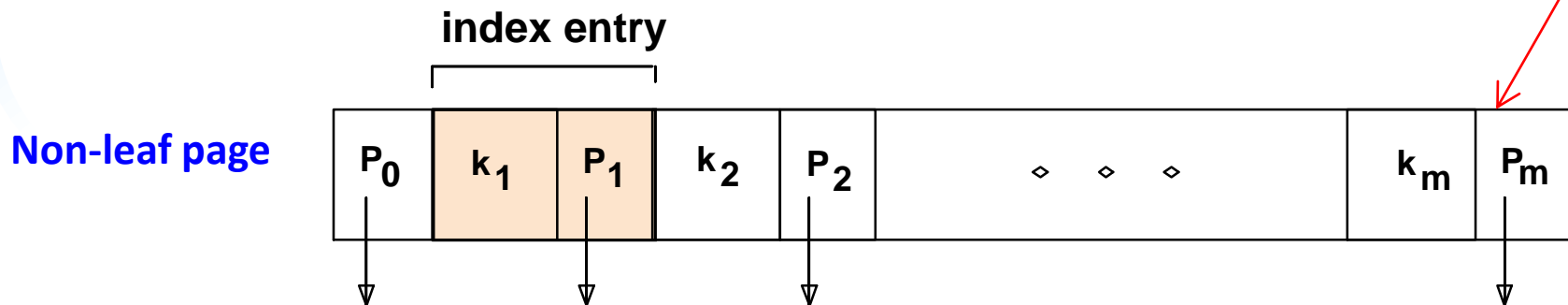
- Disadvantage of indexed-sequential files
 - performance degrades as file grows, (since many overflow blocks get created).
 - Periodic reorganization of entire file is required.
- In DBMS indexes are stored in disk-blocks
 - Indexes designed to reside on disk are very different from those designed to be in-memory, due to the access characteristics of disks!
- Would like to index the data entries:
 - Hash-based index
 - Tree-based index



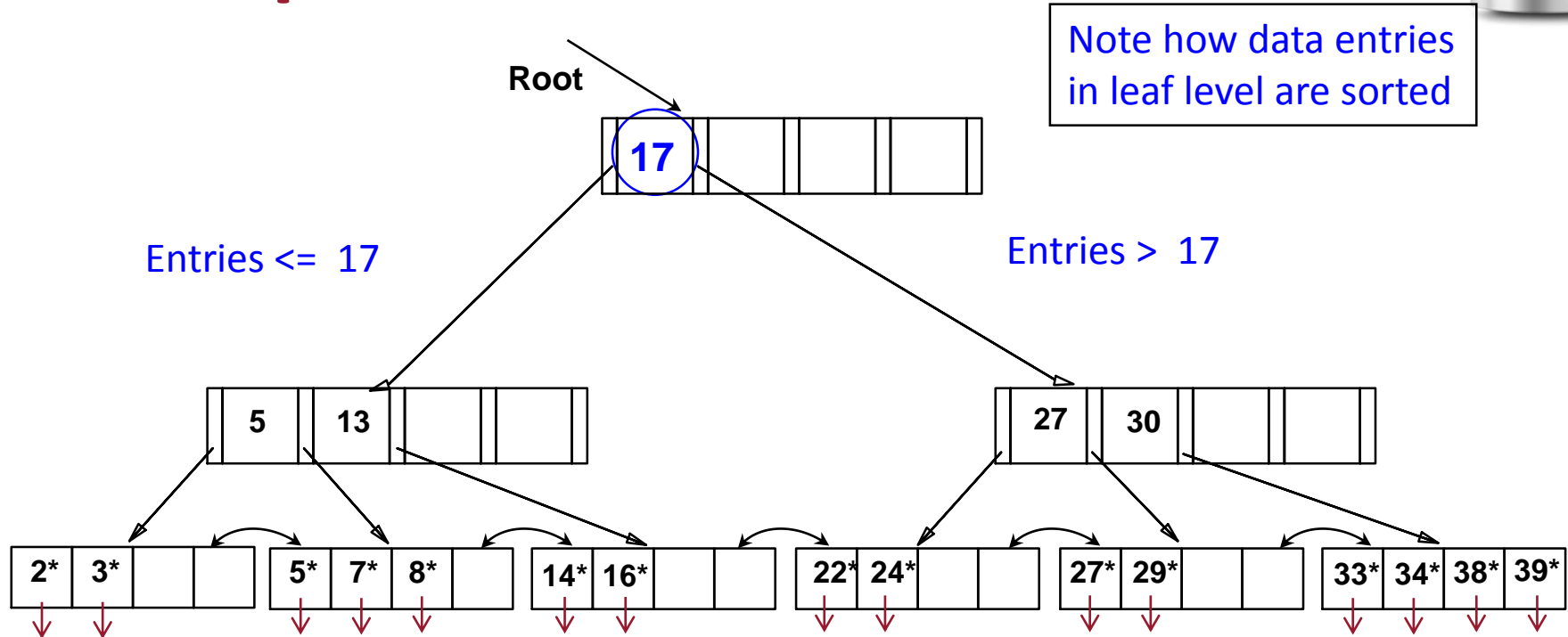
B+ Tree Indexes



- ❖ Leaf pages contain *data entries*, and are chained (prev & next)
- ❖ Non-leaf pages contain node pointers separated by search key values.



Example B+ Tree

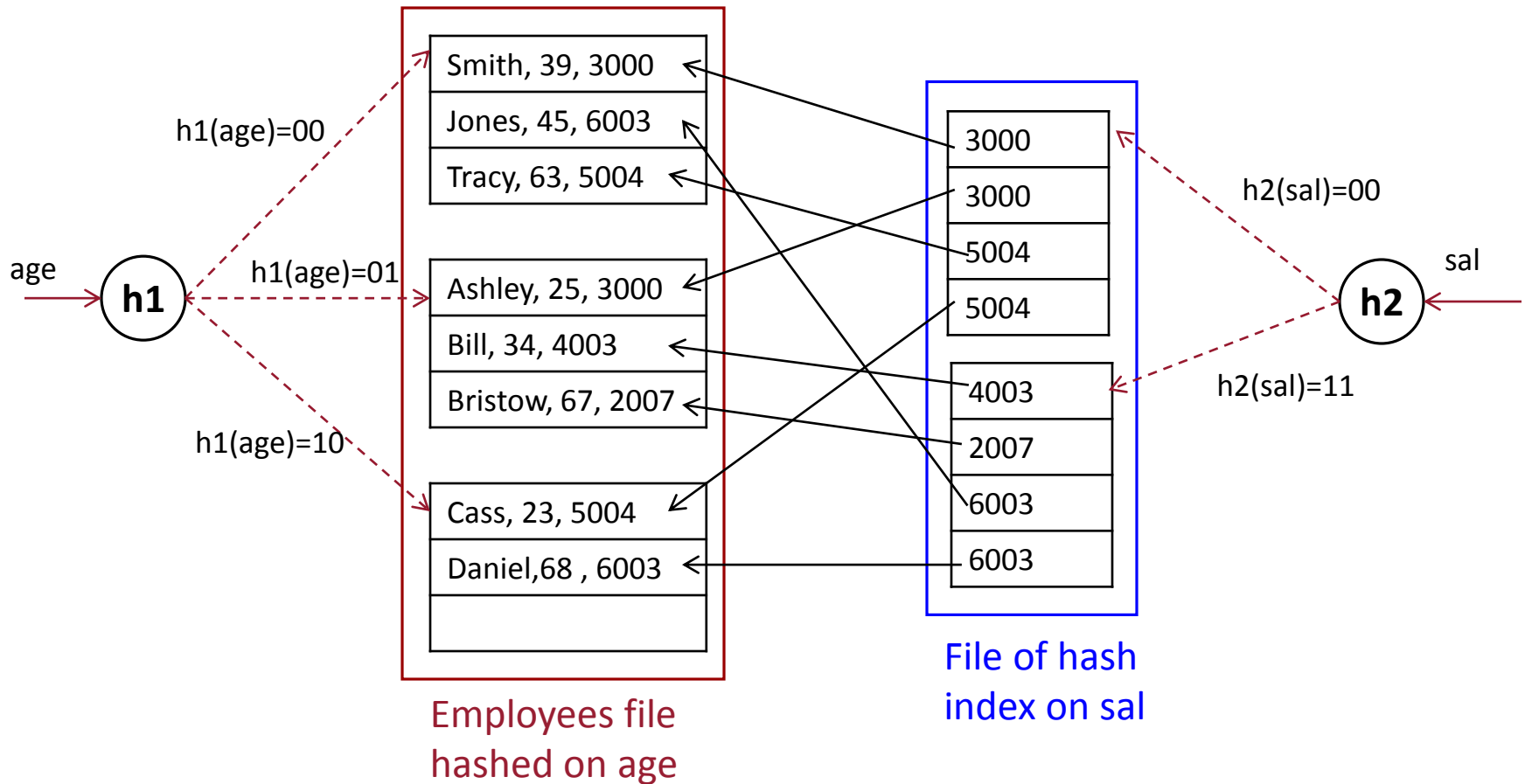


- Find 28^* , 29^* ? All $> 15^*$ and $< 30^*$
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
 - And change sometimes bubbles up the tree

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*.

Hash-Based Indexes



Index-Organized File Hashed on *age*, with Auxiliary Index on *sal*

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

- **B** : The number of data pages
- **R** : Number of records per page
- **D** : (Average) time to read or write disk page
- Average-case analysis; based on several simplistic assumptions.

✉ *Good enough to show the overall trends!*

Comparing File Organizations

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

Operations to Compare

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

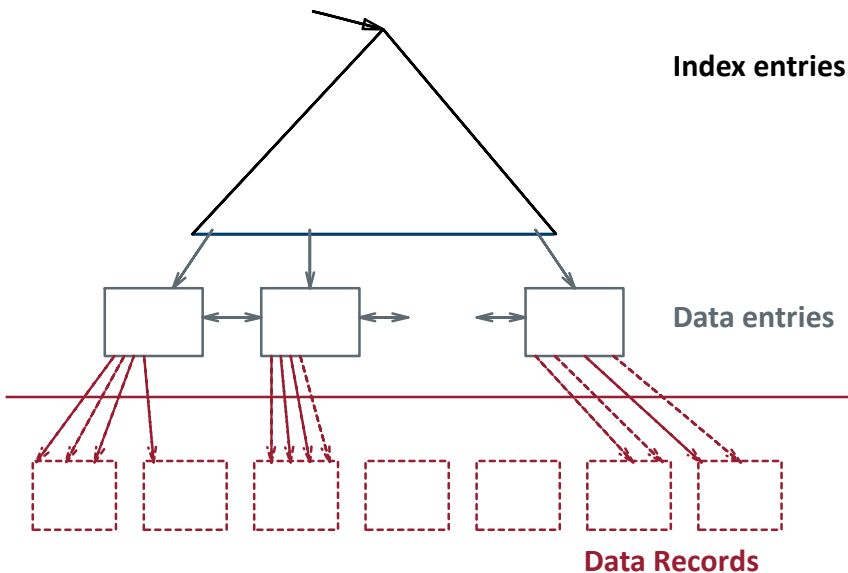
Assumptions in Our Analysis



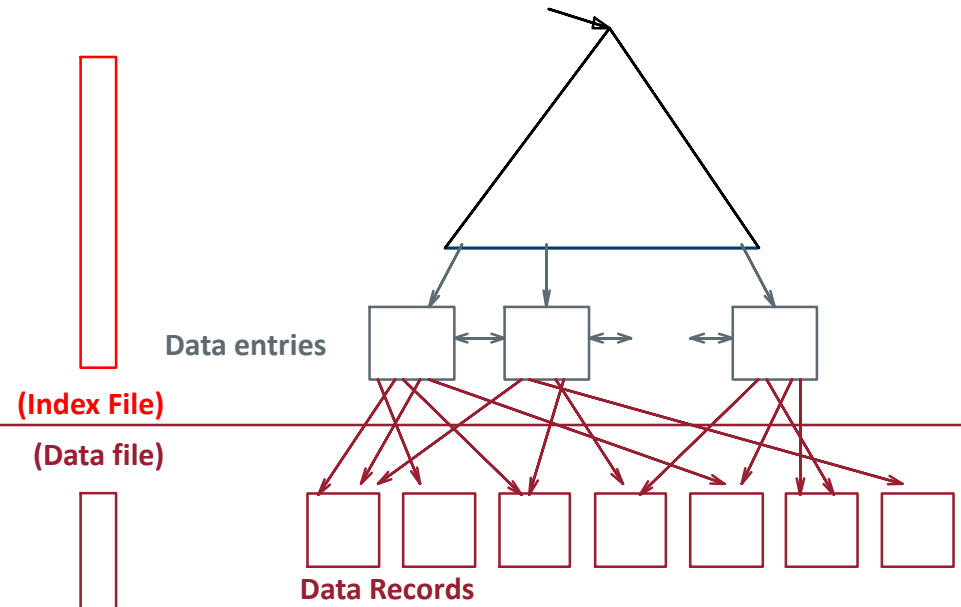
- Heap Files:
 - Equality selection on key; exactly one match.
- Sorted Files:
 - Files compacted after deletions.
- Indexes:
 - Alternative (2),(3): index data entry size = 10% size of record
 - Clustered B+ Tree file: 67% occupancy (this is typical).
 - Implies file size = 1.5 data size
 - Unclustered B+ Tree: 67% occupancy of index pages
 - Implies index size = $1.5 * 0.1 * \text{data size}$ (since index data entry size is 10% size of record)
 - B+ Tree: fan-out is F
 - B+ Tree: Assume we can directly access the first leaf page
 - Hash: No overflow buckets.
 - 80% page occupancy => File size = 1.25 data size
- Scans:
 - Leaf levels of a tree-index are chained.
 - Index data-entries plus actual file scanned for unclustered indexes.
- Range searches:
 - We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.

Clustered vs. Unclustered Index

CLUSTERED



UNCLUSTERED



Clustered = records close in index are close in data

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	$D \log_2 B$	$D(\log_2 B + \text{\# pgs with match recs})$	Search + BD	Search + BD
(3) Clustered B+ tree file					
4) Unclustered B+ tree index					
4) Unclustered Hash index					

✉ *Several assumptions underlie these (rough) estimates!*

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	$D \log_2 B$	$D(\log_2 B + \text{\# matching pages})$	Search+ BD	Search + BD
(3) Clustered B+ tree file	1.5BD (sorted)	$D \log_F 1.5B$	$D(\log_F 1.5B + \text{\# matching pages})$	Search +D	Search +D
4) Unclustered B+ Tree index	BD^* (unsorted) $4BD^{**}$ (sorted)	$D(\log_F 0.15B + 1)$	$D(\log_F 0.15B + \text{\# matching records})$	Search*** +2D	Search*** +2D
4) Unclustered Hash index	BD^* (unsorted) $4BD^{**}$ (sorted)	2D	BD^*	Search*** +2D	Search*** +2D

* Don't use index.

**Includes cost of sorting

***Search time also includes the time to fetch data block

✉ Several assumptions underlie these (rough) estimates!

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 updates in the workload:
 - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.
 - Frequency of the updates.

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.
 - Range query suggests tree index.
 - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
2. Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
 - Order of attributes is important for range queries.
 - Such indexes can sometimes enable **index-only** strategies for important queries.
 - For index-only strategies, clustering is not important!
3. 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

- B+ tree index on *E.age* can be used to get qualifying tuples.
 - How selective is the condition?
 - Should the index be clustered?
- Consider the GROUP BY query.
 - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
 - Clustered *E.dno* index may be better!
- Equality queries and duplicates:
 - A hash index on *E.hobby* helps!
 - Should the index be clustered?

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

```
SELECT E.dno, COUNT (*)  
FROM Emp E  
WHERE E.age>10  
GROUP BY E.dno
```

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

Composite Search Keys

- To retrieve Emp records with *age*=22 AND *sal*=20K,
 - An index on *<age,sal>* would be better than an index on *<age>* or an index on *<sal>*.
- If condition is: 20<*age*<30 AND 30K<*sal*<80K:
 - Clustered tree index on *<age,sal>* or *<sal,age>* is best.
- If condition is: *age*=23 AND 30K<*sal*<50K:
 - Clustered *<age,sal>* index much better than *<sal,age>* index!
- Composite indexes are larger, updated more often.

```
SELECT *
FROM Emp E
WHERE E.age = 22
AND E.sal = 20K
```

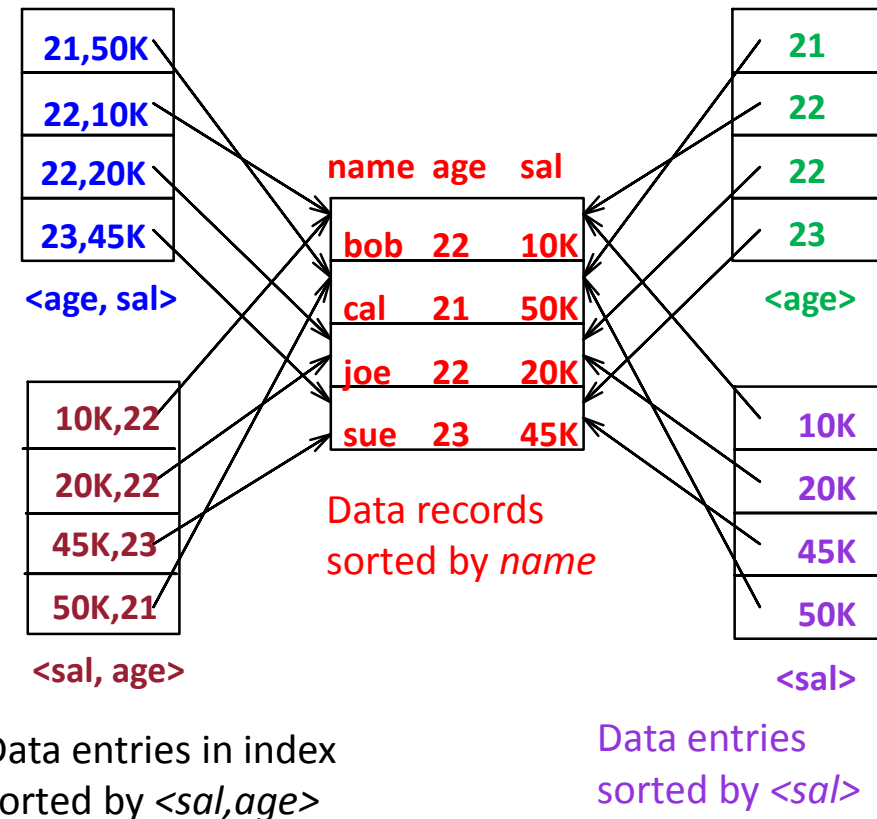
```
SELECT *
FROM Emp E
WHERE E.age > 20 AND E.age < 30
AND E.sal > 30K AND E.sal < 80K
```

```
SELECT *
FROM Emp E
WHERE E.age = 23 AND
E.sal > 30K AND E.sal < 50K
```

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 $\langle \text{sal}, \text{age} \rangle$ index:
 - age=22 and sal =10K
 - **Range query:** Some field value is not a constant. E.g.:
 - age=22 and sal > 10K

Examples of composite key indexes using lexicographic order.



Index-Only Plans

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

<E.dno>

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

<E.dno, E.sal>

Tree index!

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

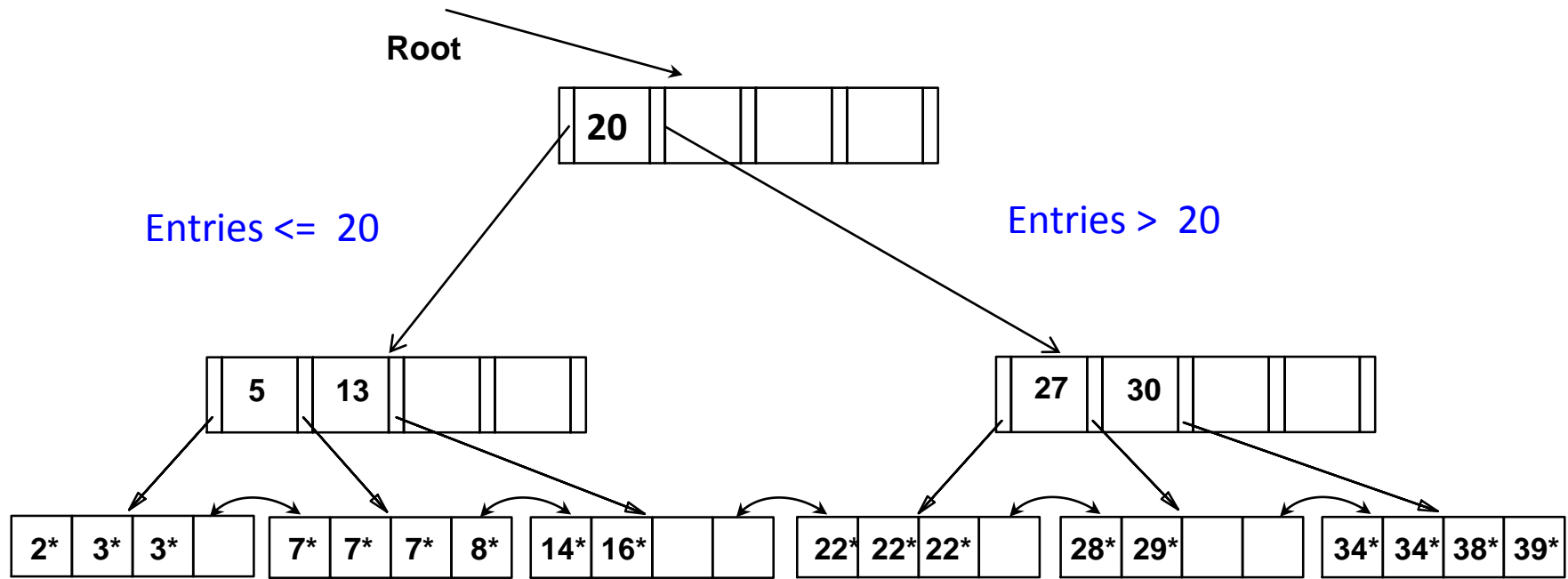
Which one is better?

*<E. age, E.sal>
or
<E.sal, E.age>*

Tree index!

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
      E.sal > 30K AND E.sal < 50K
```

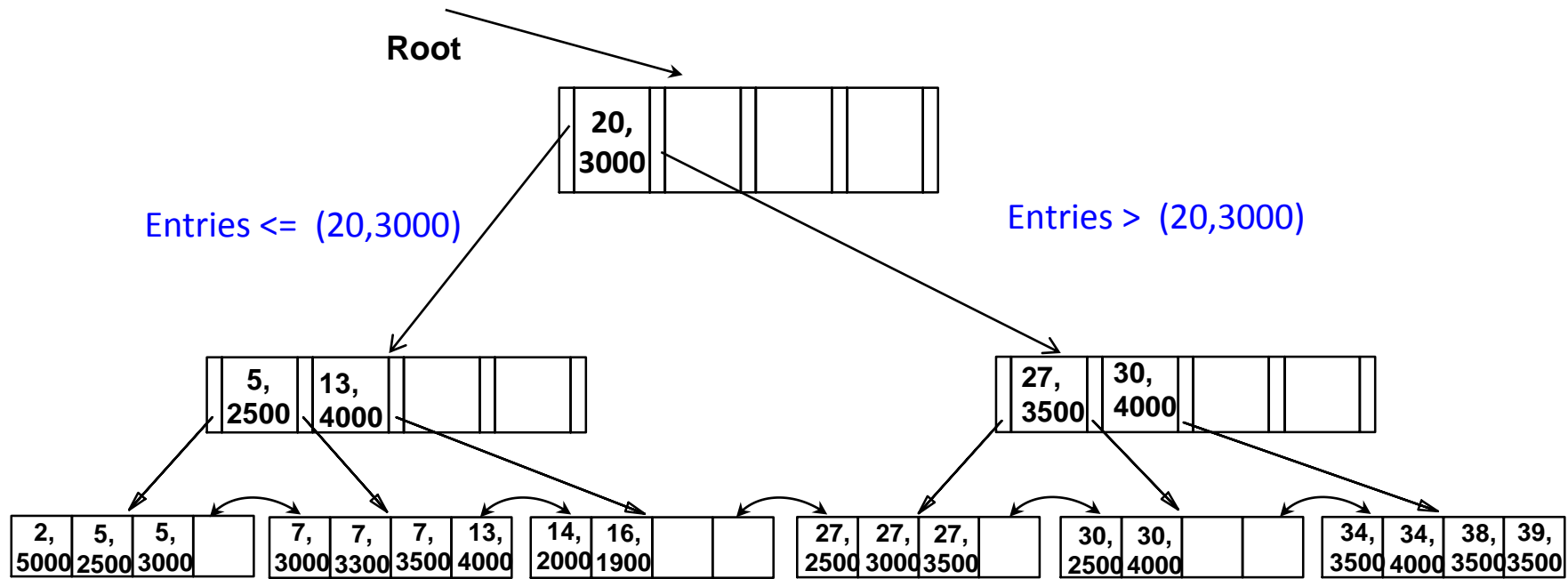
Index-Only Plans - Examples



Index on $\langle E.dno \rangle$

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

Index-Only Plans - Examples



Index on $\langle E.dno, E.sal \rangle$

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

Index-Only Plans (Contd.)

- Index-only plans are possible if we have a index on <eno,sal> or <sal,eno>.
 - Which is better?
 - What if we consider the second query?

```
SELECT E.eno, COUNT (*)  
FROM Emp E  
WHERE E.sal=3000  
GROUP BY E.eno
```

```
SELECT E.eno, COUNT (*)  
FROM Emp E  
WHERE E.sal>3000  
GROUP BY E.eno
```

Index-Only Plans (Contd.)

- Index-only plans can also be found for queries involving more than one table.

<E.dno>

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

<E.dno,E.eid>

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


Indexes cannot solve everything...

- **Watch out for inequalities!**

- `SELECT * FROM R WHERE R.A != 10;`
 - B+Tree and Hash Index won't help
 - Scanning the table is likely to be best

- **Watch out for wildcards!**

- `SELECT * FROM R. WHERE R.B LIKE 'Hello%';`
 - B+Tree can help by finding prefixes
 - Hash Index unable to help
- `SELECT * FROM R. WHERE R.B LIKE '%Hello%';`
 - Even B+Tree cannot help...

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 plus a way to quickly find entries with given key values.

Summary (Contd.)

- Data entries can be actual data records, $\langle \text{key}, \text{rid} \rangle$ pairs, or $\langle \text{key}, \text{rid-list} \rangle$ 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, 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.