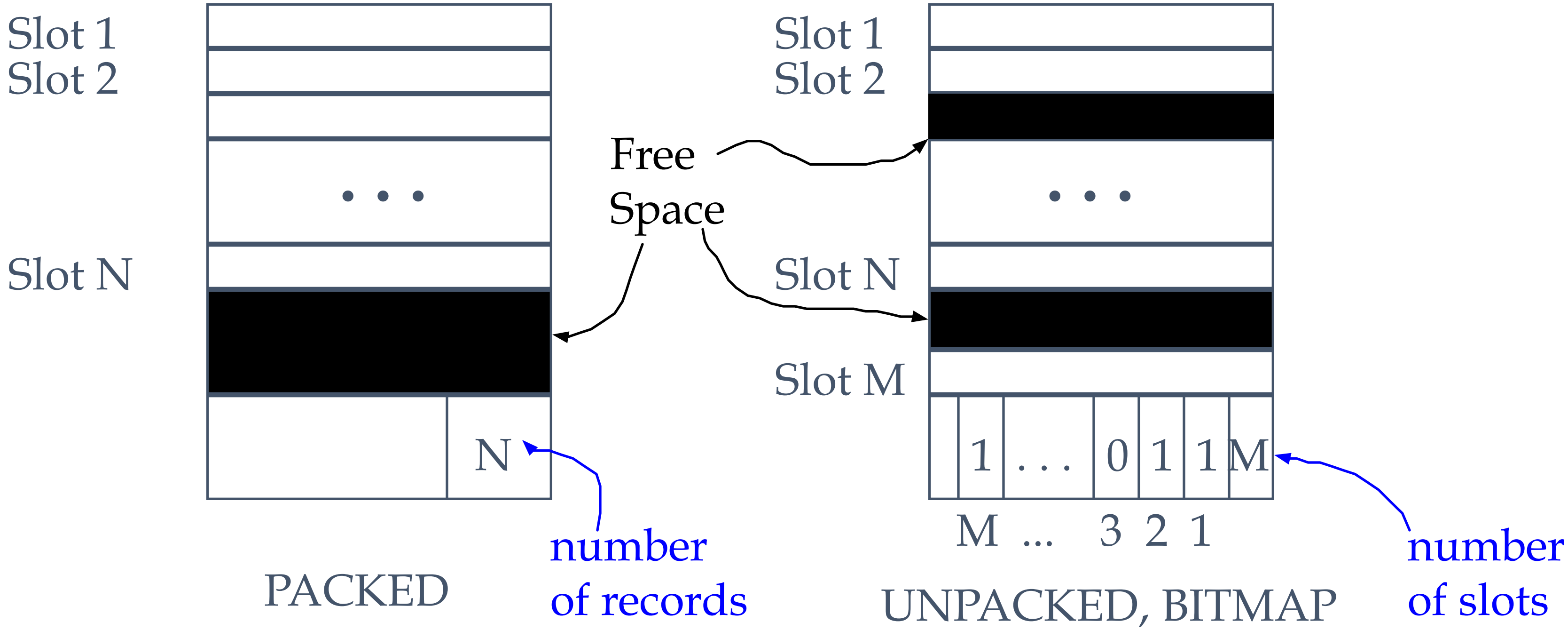
Chapter 8: Storage and Indexing

Data on external storage

- File organization: Method of arranging a file of records on external storage.
  - Record id (rid) is sufficient to physically locate record
    - Page Id and the offset on the page
- Index: data structure for finding the ids of records with given particular values faster
- Architecture: Buffer manager stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

Page Formats: Fixed Length Records



Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.

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

What is data entry k*?

- Three options depending on what level
  - Data record with key value K (actual tuple in the table)
  - <k, rid of a data record with search key value k>
    - So not the record itself the recordid (where to get the record
  - <k, list of rids of data records with a search key k>

Alternative 1 – actual data record

- Actual data record stored in index
  - 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 entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternative 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.
  - Large records take up space in the index – still have to maneuver around the portion of index structure used to direct the search, which depends on size of data entries, is much smaller than with Alternative 1.
- Alternative 3 more compact than Alternative 2, but leads to variable-sized data entries even if search keys are of fixed length.
- Extra cost for accessing data records in another file
  - Index only return rids

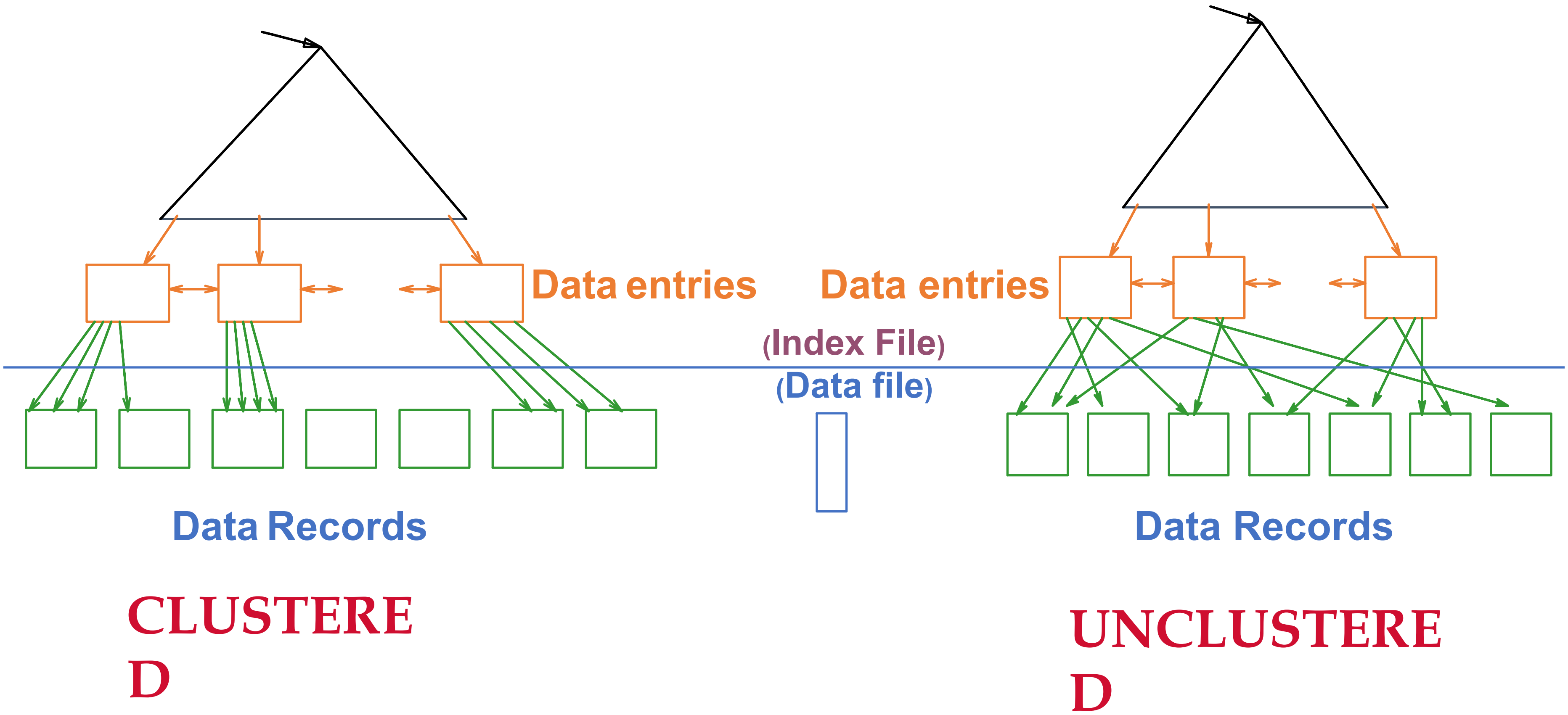
Index classification

- Primary vs. secondary: If search key contains primary key, then called primary index.
  - 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.
  - 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.
  - Cost of retrieving data records through index varies greatly based on whether index is clustered or not.

Clustered vs. Unclustered Index

- Suppose Alternative 2 is used for data entries, and that the data records are stored in a Heap file
  - To build a 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 sort order.

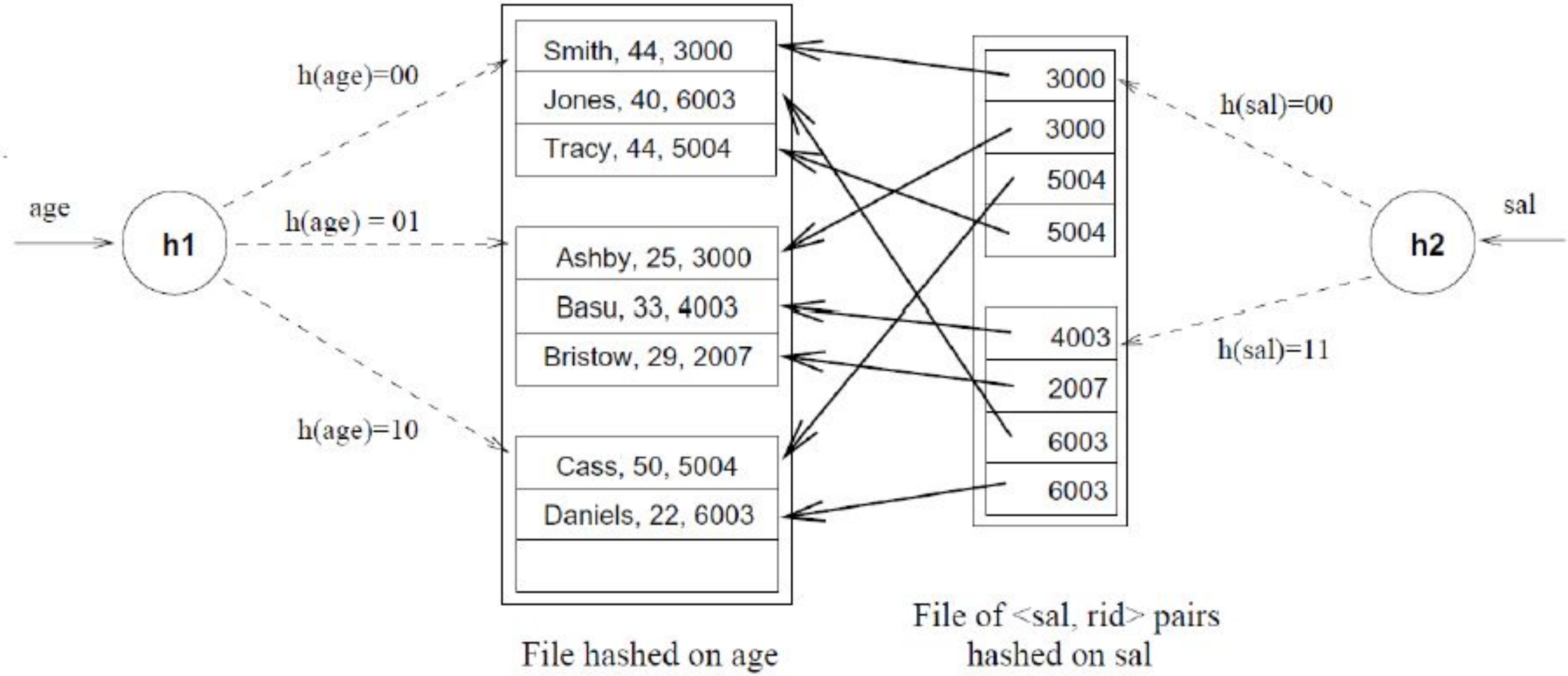
Clustered vs. Unclustered Index



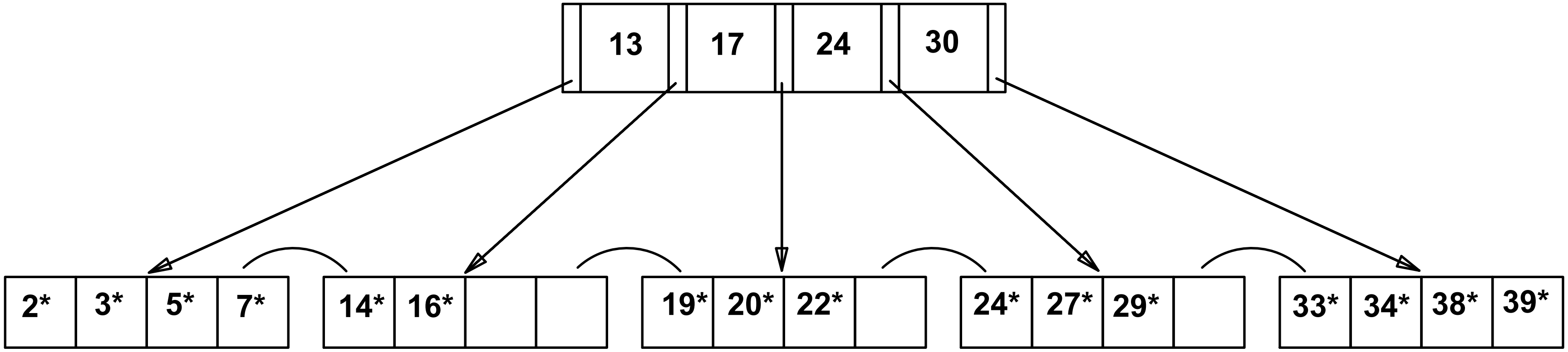
Hash-based Indexes

- Good for equality selections
  - Index is a collection of buckets. Bucket = primary page plus 0 or more overflow pages
  - Hashing function h: h(r) = bucket in which record r belongs/ h looks at the search key fields of r.
- If alternative (1) is used, the buckets contain the data records , otherwise they contain <key,rid> or <key, rid-list> pairs

Example

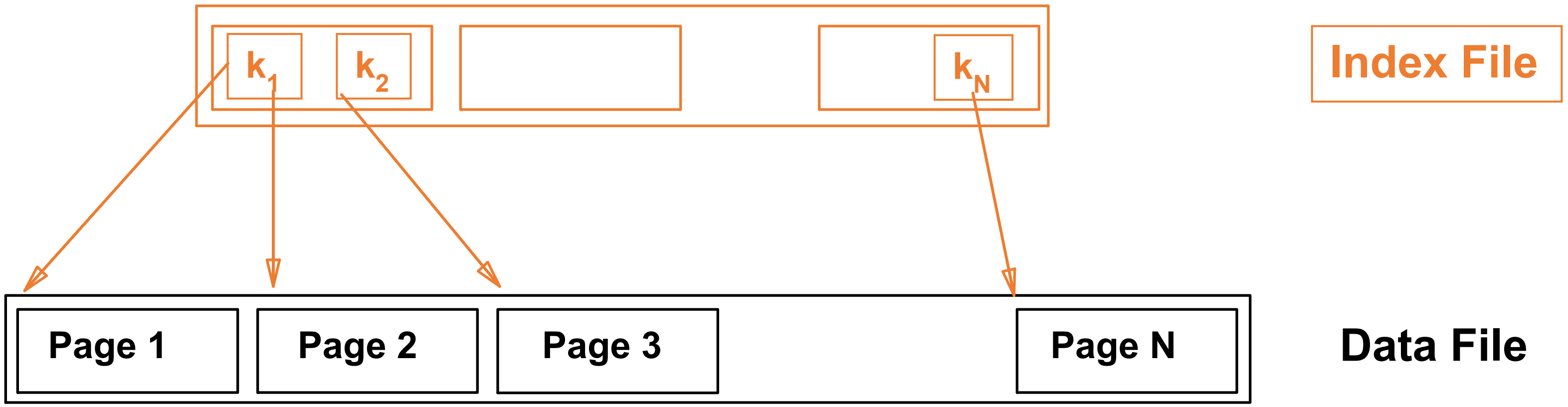

age
h1
h(age)=00
h(age) = 01
h(age)=10
Smith, 44, 3000
Jones, 40, 6003
Tracy, 44, 5004
Ashby, 25, 3000
Basu, 33, 4003
Bristow, 29, 2007
Cass, 50, 5004
Daniels, 22, 6003
3000
3000
5004
5004
4003
2007
6003
6003
h(sal)=00
h(sal)=11
sal
h2
File hashed on age
File of <sal, rid> pairs
hashed on sal

Example B+ Tree



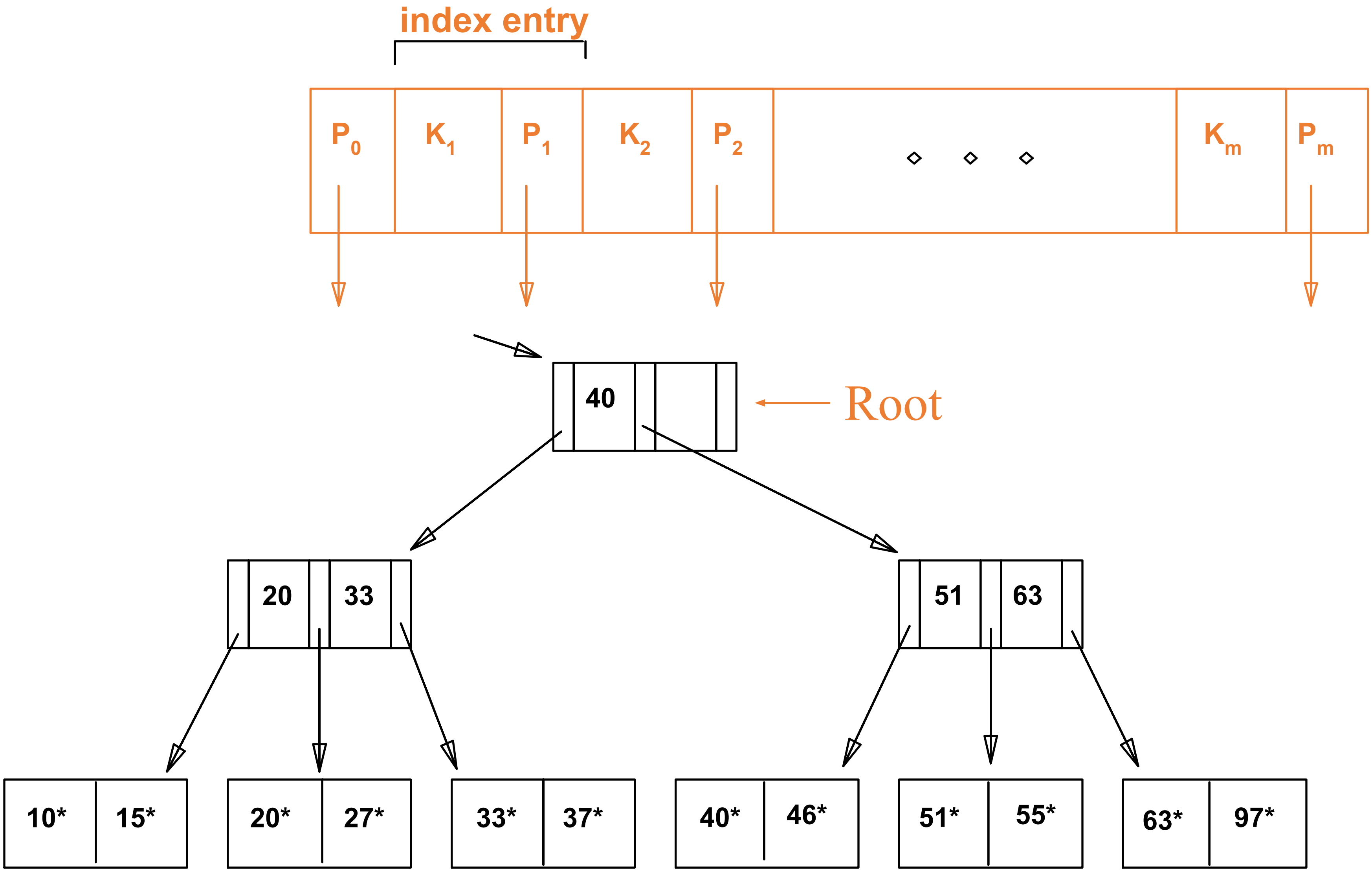
Tree-Based Indexes

- *``Find all students with grade > 92''*
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an ‘index’ file.



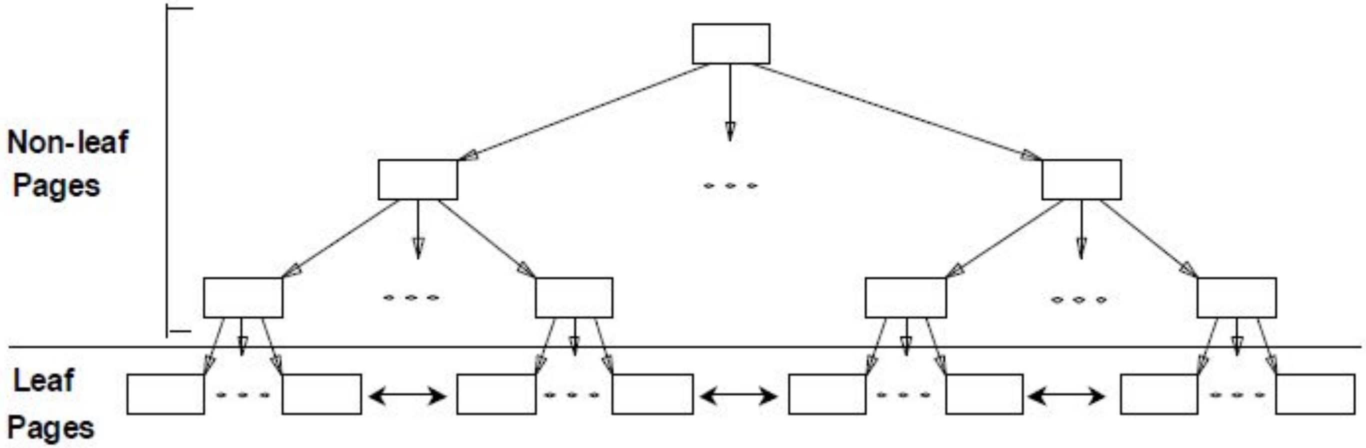
Allows performing a binary search on (smaller) index file!

Tree-Based Indexes (2)

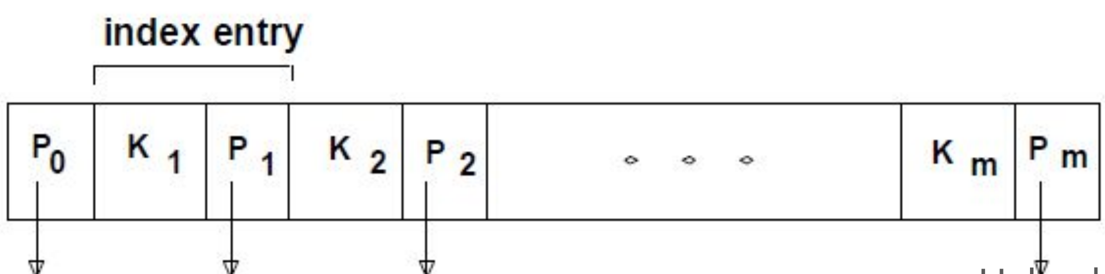


B+ Tree Indexes

B+ Tree Indexes

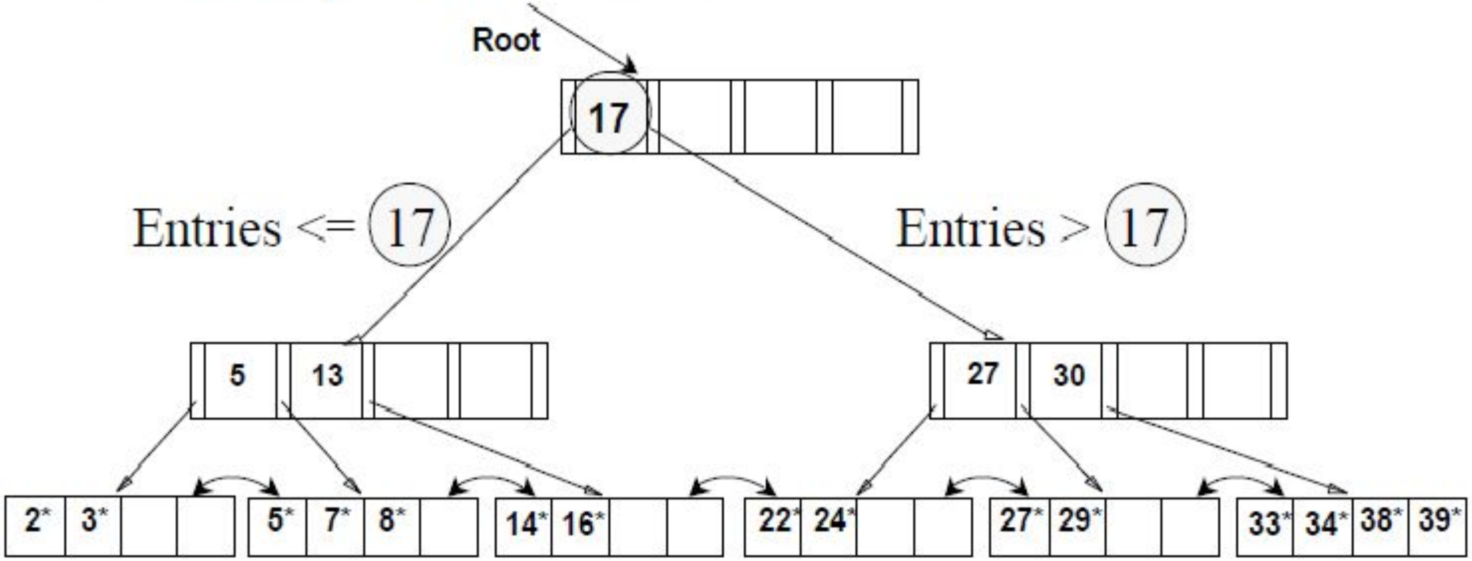


- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages contain *index entries* and direct searches:



Example: B+ Tree

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

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fan-out = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 133 pages = 1 MB
  - Level 3 = 17,689 pages = 133 MB

Cost Model Analysis

- We ignore CPU costs, for simplicity:
  - B: The number of data pages (Blocks)
  - R: Number of records per page (Records)
  - D: (Average) time to read or write a single disk page
- 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 simplifying assumptions

Far from Precise but Good enough to show the overall trends!

Comparing File Organization

- Heap files (random order; insert at eof)
- Sorted files, sorted on attributes <age, sal>
- Clustered B+ tree file, Alternative 1, search key <age, sal>
- Heap file with unclustered B + tree index on search key <age, sal>
- Heap file with unclustered hash index on search key <age, sal>

Operations to compare: Regular file (B)

- Scan: Fetch all records from disk. === B.D

- Equality search. === ½ BD
- Range selection. ==== B D
- Insert a record. ===== D + D = 2D
- Delete a record ==== search + D

Operations to compare: Sorted file (B)

- Scan: Fetch all records from disk. === B.D

- Equality search. === Binary search = D *Log_2 B
- Range selection. ==== D * Log_2B + matches
- Insert a record. ===== read half file, write it in different location

  Search + 1/2BD + 1/2BD = Search + BD
- Delete a record ==== Search + BD

Assumptions for the File Organizations

- Heap Files:
  - Equality selection on key; exactly one match.
- Sorted Files:
  - Files compacted after deletions.
- Indexes:
  - Alternatives 2, 3: data entry size = 10% of record size
- Tree: 67% occupancy (AUC for 1 std dev.).
  - Implies file size = 1.5 data size
- Hash: No overflow buckets.
  - 80% page occupancy => File size = 1.25 data size

Assumptions for Operations

- Scans:
  - Leaf levels of a tree-index are chained.
  - Need to scan the 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.
    - Why can't we use hash index?

Heap File – not sorted, no index

- **Scan** – need to read all records
  - Number of data pages B X Time to read a page D **BD**
- **Equality search**
  - On average need to search ½ the file to find a random record
  - ½ (number of data pages B X time to do a read D **) .5BD**
- **Range search**
  - Data not sorted so have to read all records to make sure you get them all
  - Number of data pages B X Time to read a page D **BD**
- **Insert a record**
  - 2 I/O operations: read the page then write the page **2D**
- **Delete a record**
  - **1 write plus the search to the current page search + D**

Sorted file – Data records sorted

- **Scan** – need to read all records
  - Number of data pages B X Time to read a page D **BD**
- **Equality search**
  - Use a binary search to locate first page to satisfy criterion
  - average Log2B reads to locate random record X cost of a read **Dlog2B**
- **Range search**
  - Use a binary search to locate first page to satisfy criterion **Dlog2B**
  - Also need a read for every other page that satisfies the criterion **Dlog2B + # matching pages**
- **Insert a record**
  - Search to the page for the insertion + **BD**
- **Delete a record**
  - Search to the page for the deletion + **BD**

Clustered file

- **Scan** – need to read all records - typically more pages since only 67% occupancy (1.5)
  - 1.5 X Number of original data pages B X Time to read a page D **1.5BD**
- **Equality search**
  - Find first leaf page to satisfy criterion in logF1.5B
  - Number of disk reads **logF1.5B X Time to read page D**
- **Range search**
  - Find first page to satisfy criterion in logF1.5B
  - Subsequent leaf nodes are read until you hit a record not satisfying the condition **LogF1.5B + # matching pages X time to read a page D**
- **Insert a record**
  - Search to the page for the insertion + **BD**
- **Delete a record**
  - Search to the page for the deletion + **BD**

Unclustered file – tree index

- **Scan** – need to read all leaf pages - typically more pages since only 67% occupancy (1.5); but smaller data entry in index .1(1.5) = .15B
  - Read all data pages cost = BD(R + .15) Expensive!
- **Equality search**
  - Find first leaf page to satisfy criterion in logF.15B
  - Number of disk reads **(1 + logF.15B) X Time to read page D**
- **Range search**
  - Find first page to satisfy criterion in logF.15B
  - Subsequent leaf nodes are read until you hit a record not satisfying the condition **D(LogF.15B + # matching pages)**
- **Insert a record**
  - Insert the data record in the file **2D**
  - Find insertion spot in index **DLogF.15B, do insertion D => D(3 + LogF.15B)**
- **Delete a record**
  - Search to the page for the deletion + **2D (index + data write)**

Unclustered file – hash index

- **Scan** – need to read all leaf pages - typically pages only 80% occupancy (1.25); but smaller data entry in index .1(1.25) = .125B
  - Read all data pages for every record cost = RBD
  - Read index = .125BD Total = RDB + .125BD Expensive!
- **Equality search**
  - Find read index page **D**
  - Read data page **D**
- **Range search – no help from index since hashing value**
  - Read entire heap file **BD**
- **Insert a record**
  - Read , write data record **2D**
  - Read, write index **2D** **Total cost (4D)**
- **Delete a record**
  - Search to the page for the deletion + **2D (index + data write)**

Cost of Operations (I/O only)

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

Choosing an index

- 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 or tree?
- Access method: index-only, index + data file

Choice of indexes

- 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.
- Must understand how a DBMS evaluates queries and creates query evaluation plans.
- 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 guideline

- Attributes in **WHERE clause** are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Matches big = selectivity low = clustered tree
    - Matches a few = selectivity high = unclustered tree
- Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.: most selective first
- Such indexes can sometimes enable index-only strategies for important queries: when only indexed attributes are needed.
- For index-only strategies, clustering is not important.
  - Try to choose indexes that benefit many queries.
  - Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Examples of cluster index

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
- Is the index 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:
- Clustering on E.hobby helps!

```
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'
```

Indexes with composite key search

- Composite Search Keys: Search on a combination of fields.
  - Equality query: **Every** field value is equal to a constant. 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.
  - Lexicographic order, or Spatial order

age	sal
21	80
22	10
24	20
25	75

Examples of composite key
Index using
Lexicographic order.

age
21
22
24
25

Name	Age	Sal
Bob	22	10
Cal	21	80
Joe	24	20
Sue	25	75

Data records
Sorted by
name

Sal	Age
80	21
75	25
20	24
10	22

Date entries in
index
Sorted by

sal
10
20
75
80

Date entries
Sorted by
<sal>

Composite Search Keys

- To retrieve Emp records with age=30 AND sal=4000, an index on <age,sal> would be better than an index on age alone or an index on sal.
  - Choice of index key orthogonal to clustering etc.
- If condition is 20<age<30 AND 3000<sal<5000:
  - Clustered tree index on <age,sal> or <sal,age> is best.
- If condition is age=30 AND 3000<sal<5000:
  - Clustered <age,sal> index much better than <sal,age>
- index.
- Composite indexes are larger, updated more often.

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 index is available.

<edno>

<E.dno,E.eid> Tree index

<E.dno>

<E.dno,E.sal> Tree index

.

- SELECT D.mgr FROM Dept D, Emp E WHERE D.dno=E.dno
- SELECT D.mgr, E.eid FROM Dept D, Emp E WHERE D.dno=E.dno
- SELECT E.dno, COUNT(*) FROM Emp E GROUP BY E.dno
- SELECT E.dno, MIN(E.sal) FROM Emp E GROUP BY E.dno

When to use index-only plans?

- Index-only plans are possible if the key is <dno,age> or we have a tree index with key <age,dno>
  - Which is better?
  - What if we consider the second query?

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

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

Summary: File Organization

- CREATE INDEX ON TABLE student(sid);

- Many alternatives file organizations exists, each appropriate in some situations
- 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 search key values

Summary: Index

- 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: Workload to Index

- Understanding the nature of the workload and performance goals essential to developing a good design.
  - What are the important queries and updates?
  - What attributes and 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.

Example 8.11

Consider the following relations:

Emp(*eid: integer*, *ename: varchar, sal: integer, age: integer, did: integer)*

Dept(*did: integer*, *budget: integer, floor: integer, mgr eid: integer)*

Salaries range from $10,000 to $100,000, ages vary from 20 to 80, each department has about five employees on average, there are 10 floors, and budgets vary from $10,000 to $1 million. You can assume uniform distributions of values.

Which of the listed index choices would you choose to speed up the query? If your database system does not consider index-only plans (i.e., data records are always retrieved even if enough information is available in the index entry), how would your answer change? Explain briefly.

1. Query: *Print ename, age, and sal for all employees.*

(a) Clustered hash index on *ename, age, sal fields of Emp.*

(b) Unclustered hash index on *ename, age, sal fields of Emp.*

(c) Clustered B+ tree index on *ename, age, sal fields of Emp.*

(d) Unclustered hash index on *eid, did fields of Emp.*

(e) No index.

Emp(*<u>eid: integer</u>, ename: varchar, sal: integer, age: integer, did: integer)*

Dept(*<u>did: integer</u>, budget: integer, floor: integer, mgr eid: integer)*

Salaries range from $10,000 to $100,000, ages vary from 20 to 80, each department has about five employees on average, there are 10 floors, and budgets vary from $10,000 to $1 million. You can assume uniform distributions of values.

Query: *Find the dids of departments that are on the 10th floor and have a budget of less than $15,000.*

(a) Clustered hash index on the *floor field of Dept.*

(b) Unclustered hash index on the *floor field of Dept.*

(c) Clustered B+ tree index on *floor, budget fields of Dept.*

(d) Clustered B+ tree index on the *budget field of Dept.*

(e) Unclustered B+ on budget,floor, did