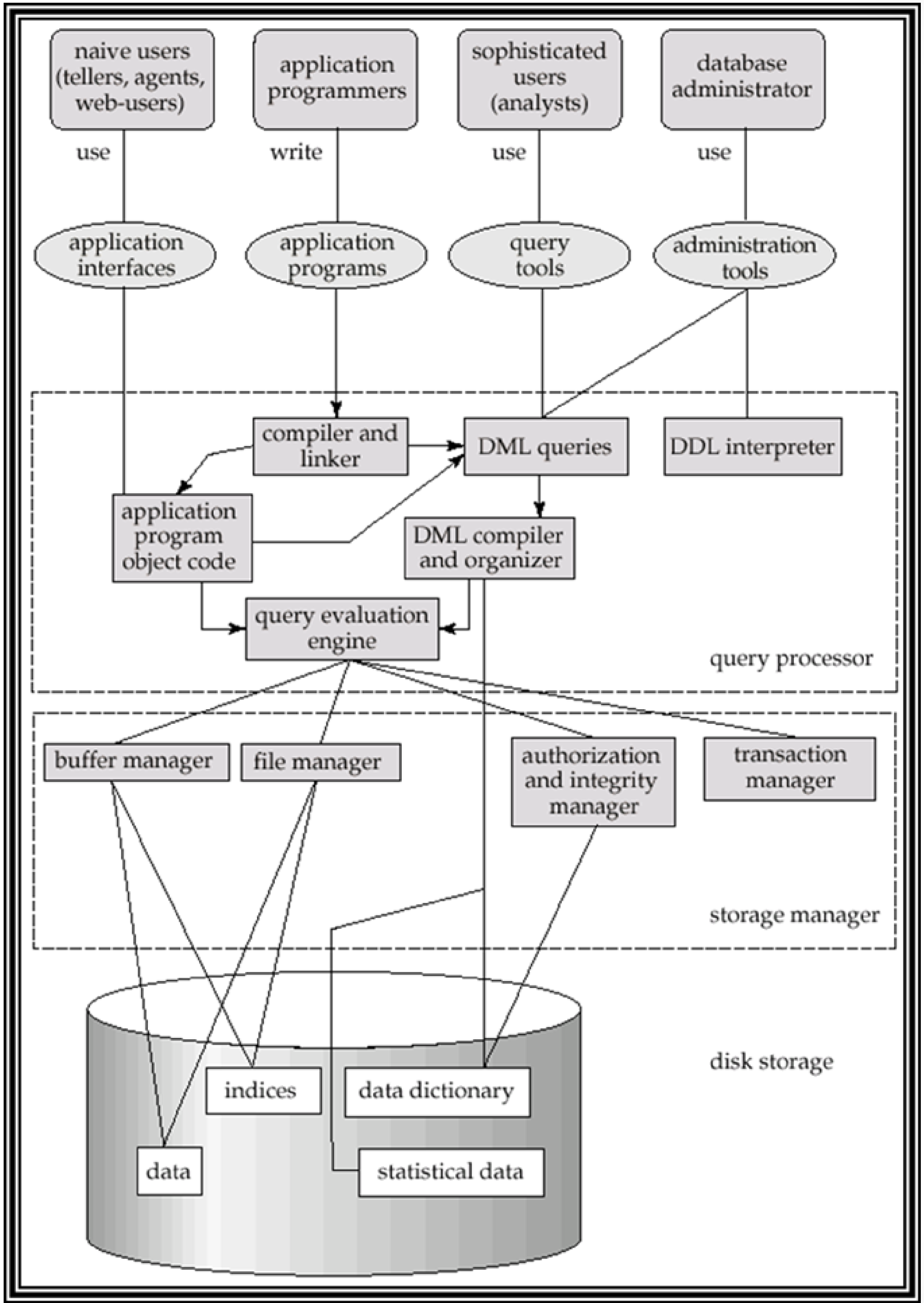
File Storage and Indexing

Lesson 13
CS 3200
Kathleen Durant PhD

Today's Topics

- Overview of data flow: External storage to RAM
- File organizations available
  - Effects on DBMS performance
- Introduction to indexes
  - Clustered vs. Unclustered
- Model for evaluating the cost of DB operations for the different file organizations
- Methods available for improving system performance
  - Indexes and when to use them or not to use them (while evaluating a query)

What topics have we covered so far?



SQL

Interface

Properties of DBMS

Glossed over Translating transactions to database actions

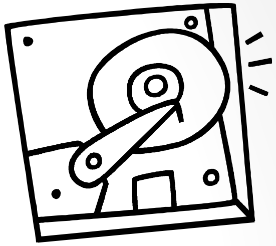
Transactions

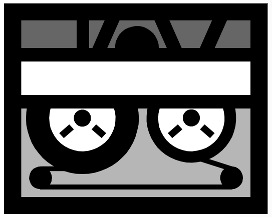
Page buffer

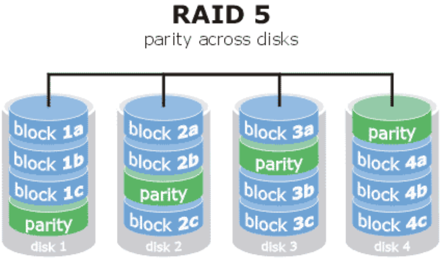
Recovery

Where can you store a database?

- 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
- Flash memory: Starting to replace disks due to much faster random access
  - Writes still slow, size often too small for DB applications
- Arrays of disks
  - Cover in later chapter







Why care about the data storage mechanism?

- DB performance depends on the time it takes to get the data from the storage system
  - I/O operations are slow

- It's all about expectations
  - If you are reading lots of data then it's OK to take a while
  - If you are reading a small amount of data it should be quick

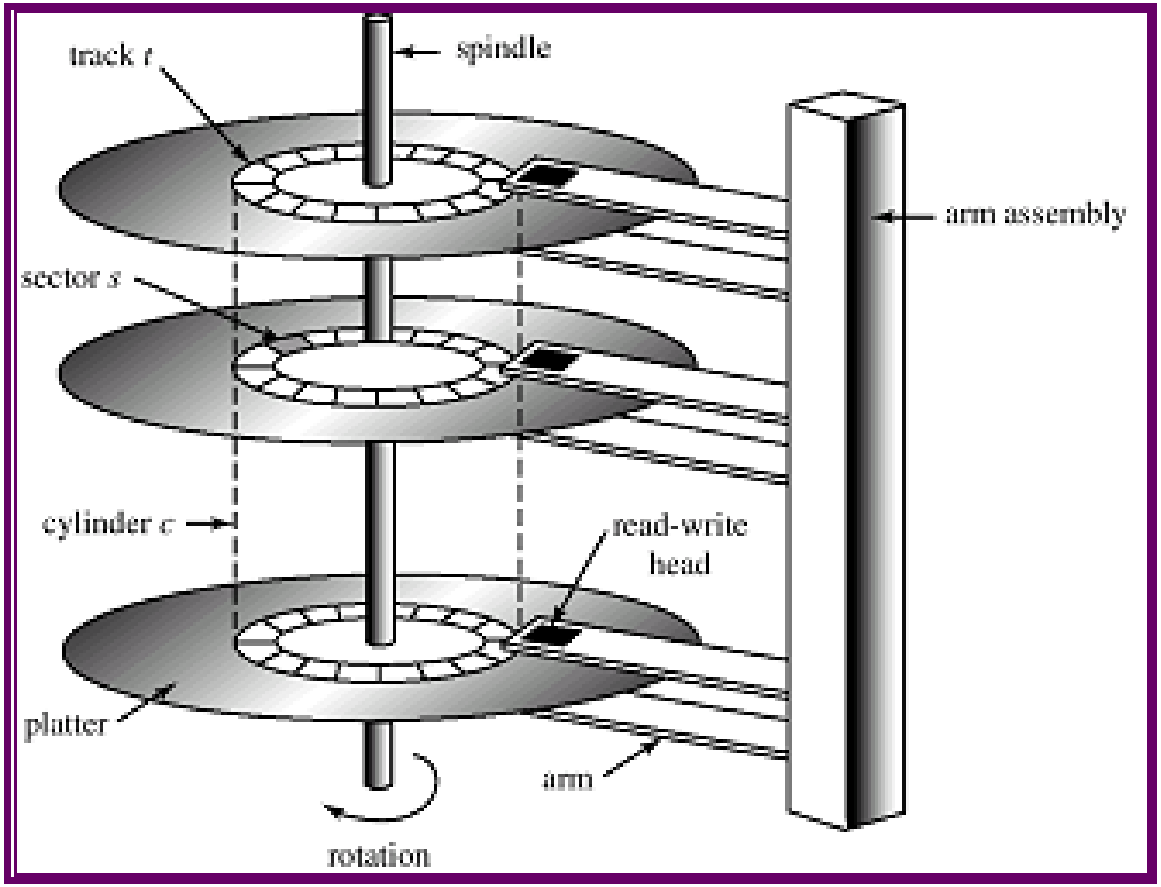
Goal: store the data in an order that will make it easy to find / identify a particular record(s)

Data stored 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.

Components of a disk

- Platters spin
  - E.g., 10K rpm
- Arm assembly is moved in or out to position a head on a desired track.
- Tracks under heads make a cylinder.
- Only one head reads or writes at any one time.
- Block size is a multiple of a sector size (which is fixed amount of data).
  - 512 bytes (old), 4096 bytes (new)



Accessing a disk page

- Time to access (read/write) a disk block:
  - **Seek time** (moving arms to position disk head on track)
  - **Rotational delay** (waiting for block to rotate under head)
  - **Transfer time** (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time typically a little below 9msec (consumer disks)
  - Rotational delay around 4msec on average (7.2K rpm disk)
  - Transfer rate disk-to-buffer ~70MB/sec (sustained)
- Key to lower I/O cost: reduce seek/rotation delays.
  - Hardware vs. software solutions ?

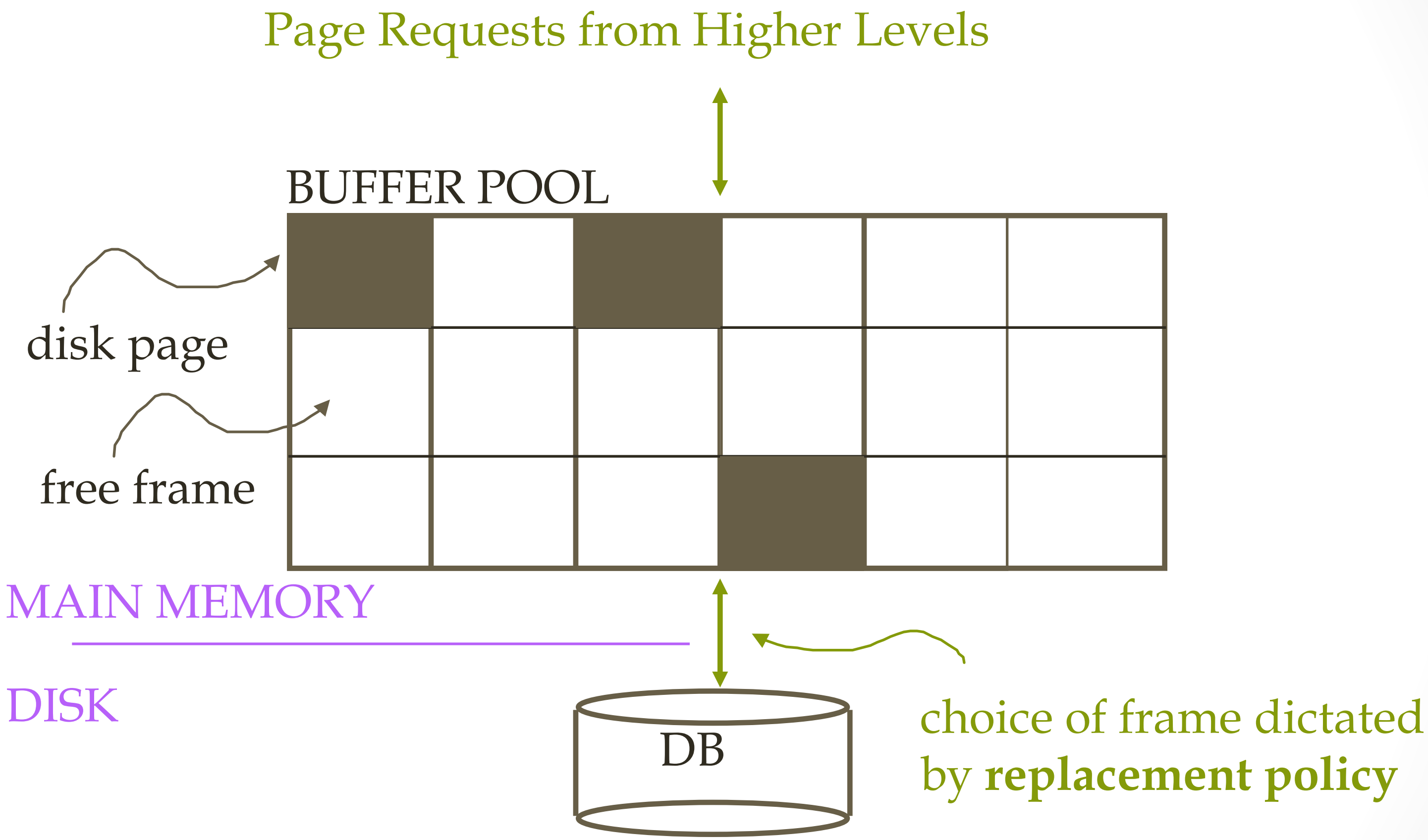
Arranging Pages on Disk

- *`Next'* block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Blocks in a file should be arranged sequentially on disk (by `next'), to minimize seek and rotational delay.
- For a sequential scan, *pre-fetching* several pages at a time is a big win!

Disk Space Management

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
    - allocate/de-allocate a page
    - read/write a page
- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk. Higher levels don’t need to know how this is done, or how free space is managed.

Buffer Management in a DBMS



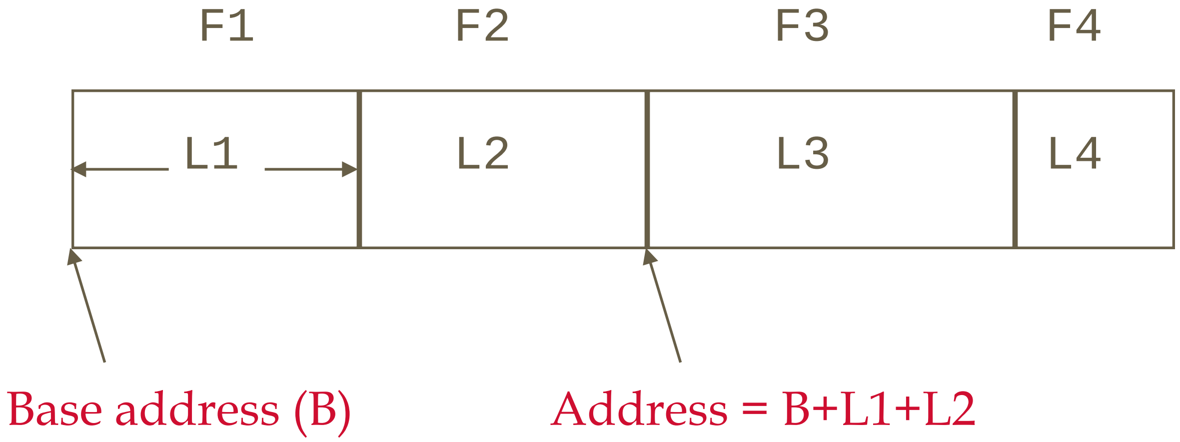
choice of frame dictated by **replacement policy**

- *Data must be in RAM for DBMS to operate on it!*
- *Table of <frame#, pageid> pairs is maintained.*

File structure types

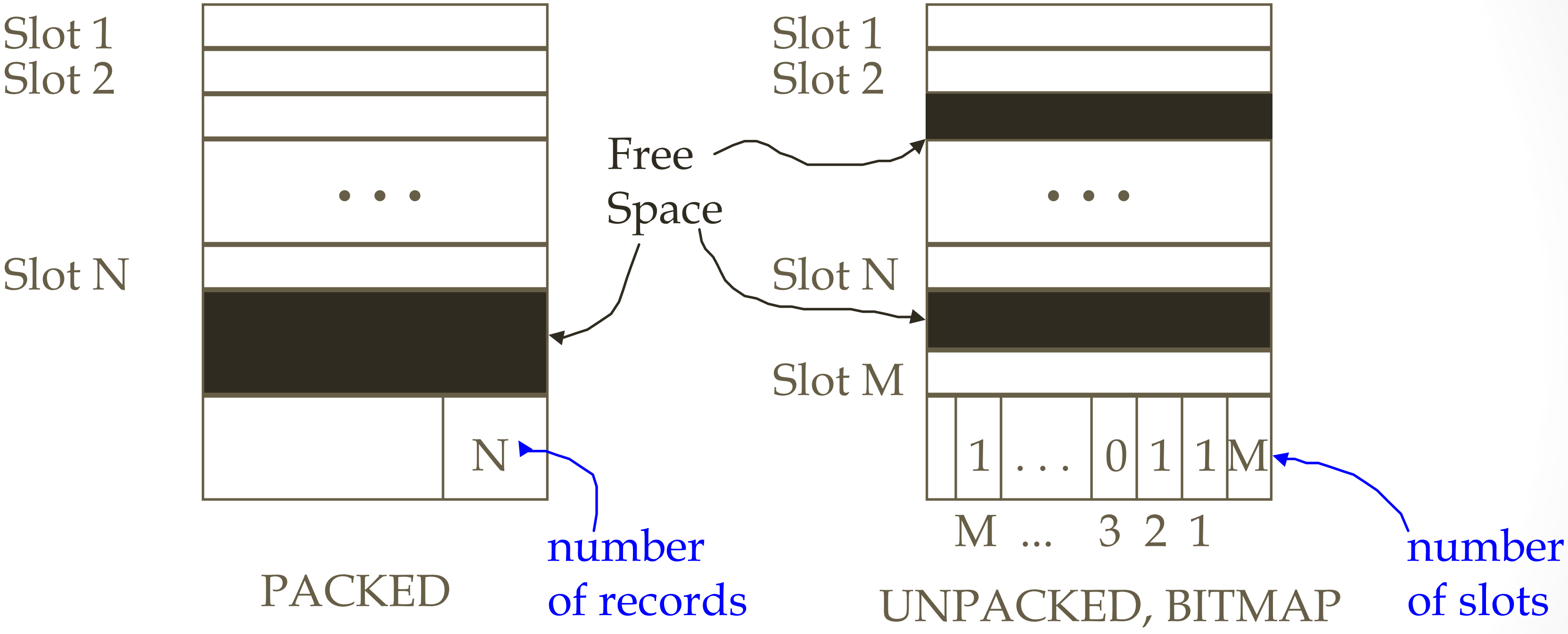
- Heap (random order) files
  - Suitable when typical access is a file scan retrieving all records.
- Sorted Files
  - Best if records must be retrieved in some order, or only a `range’ of records is needed.
- Indexes = data structures to organize records via trees or hashing.
  - Like sorted files, 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.

Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.

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* (Index)?

- Three options depending on what level
  1. Data record with key value K (actual tuple in the table)
  2. <k, rid of a data record with search key value k>
     - So not the record itself the record id (rid , where to find the data record on disk)
  3. <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

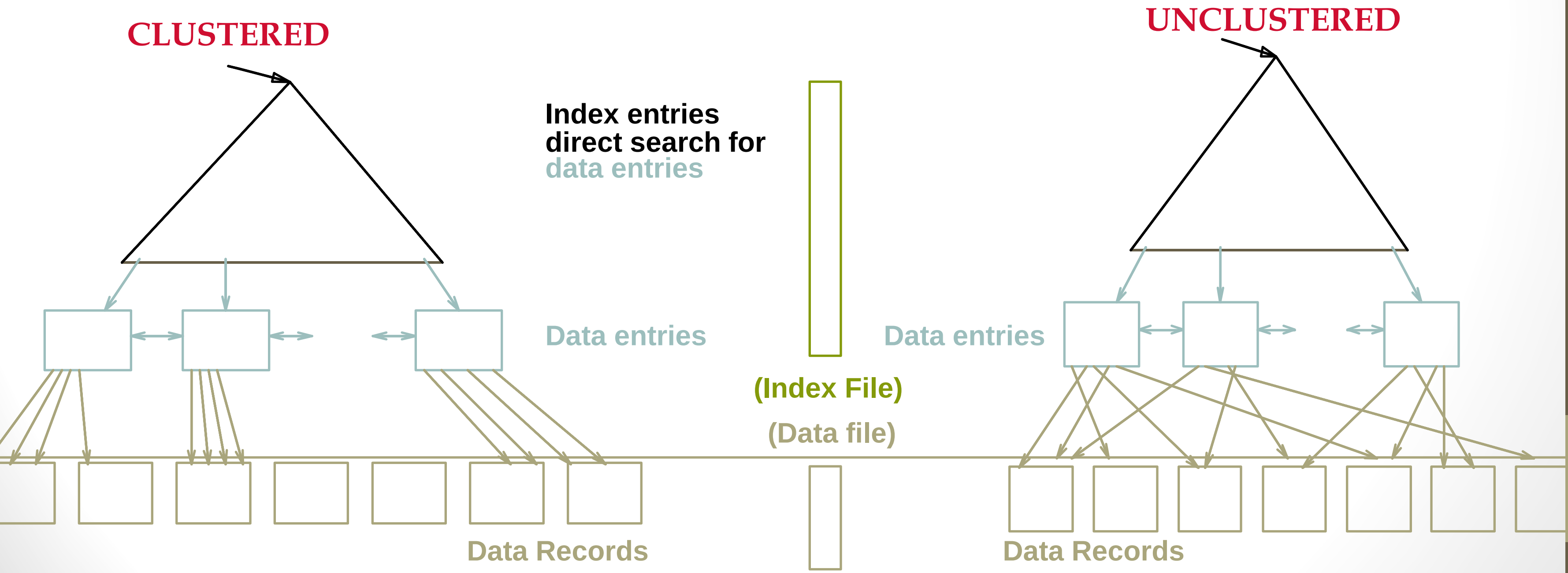
- Primary vs. secondary: If search key contains primary key, then index called primary index.
  - Unique index: Search key contains a candidate key.
- Clustered vs. unclustered: If order of data records is the same as, or `close to', order of data entries, then called clustered index.
  - Alternative 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
    - Leave 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

- Suppose that 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 recs is `close to’, but not identical to, the sort order.)



CLUSTERED

UNCLUSTERED

Index entries
direct search for
data entries

Data entries

Data entries

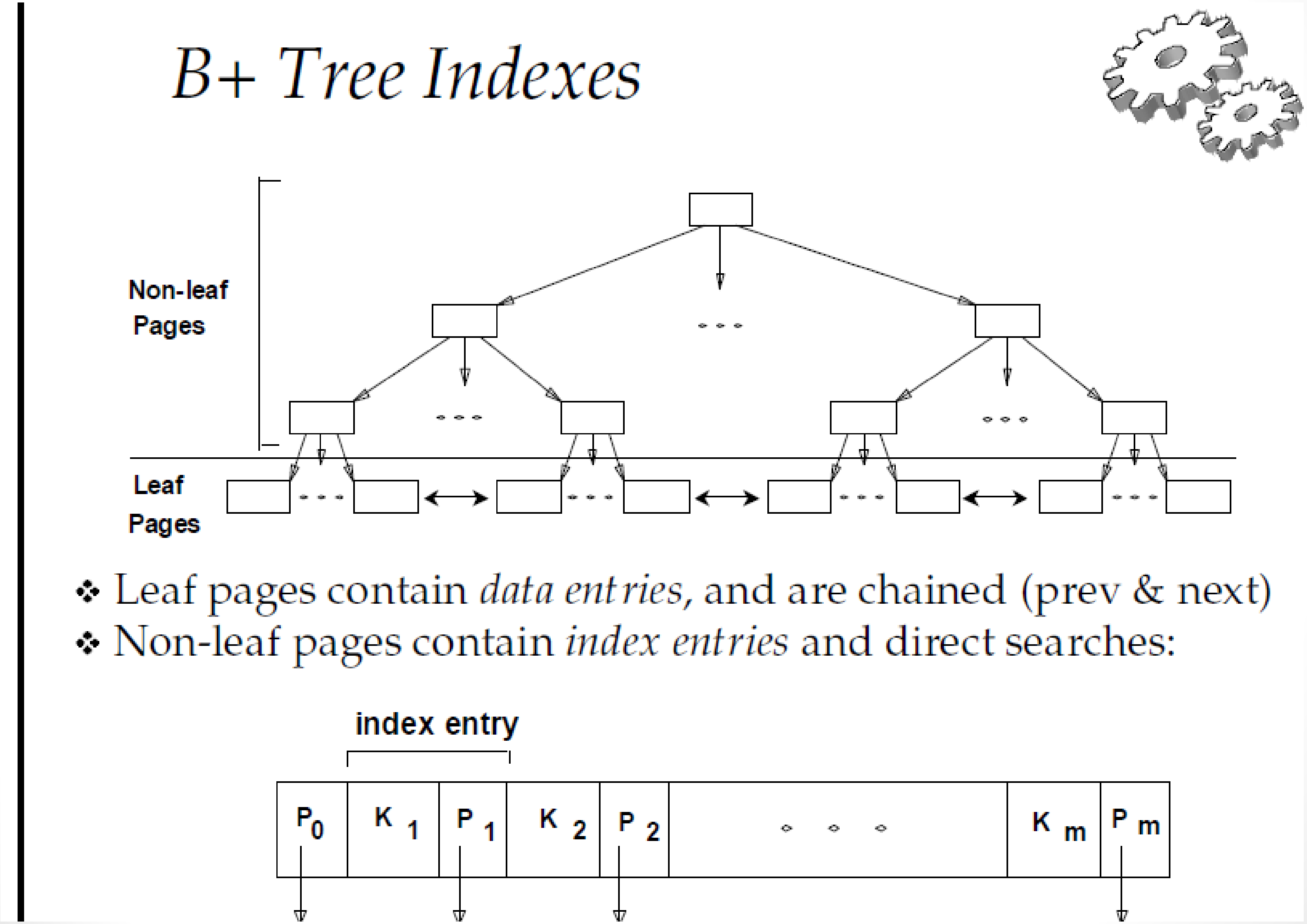
(Index File)

(Data file)

Data Records

Data Records

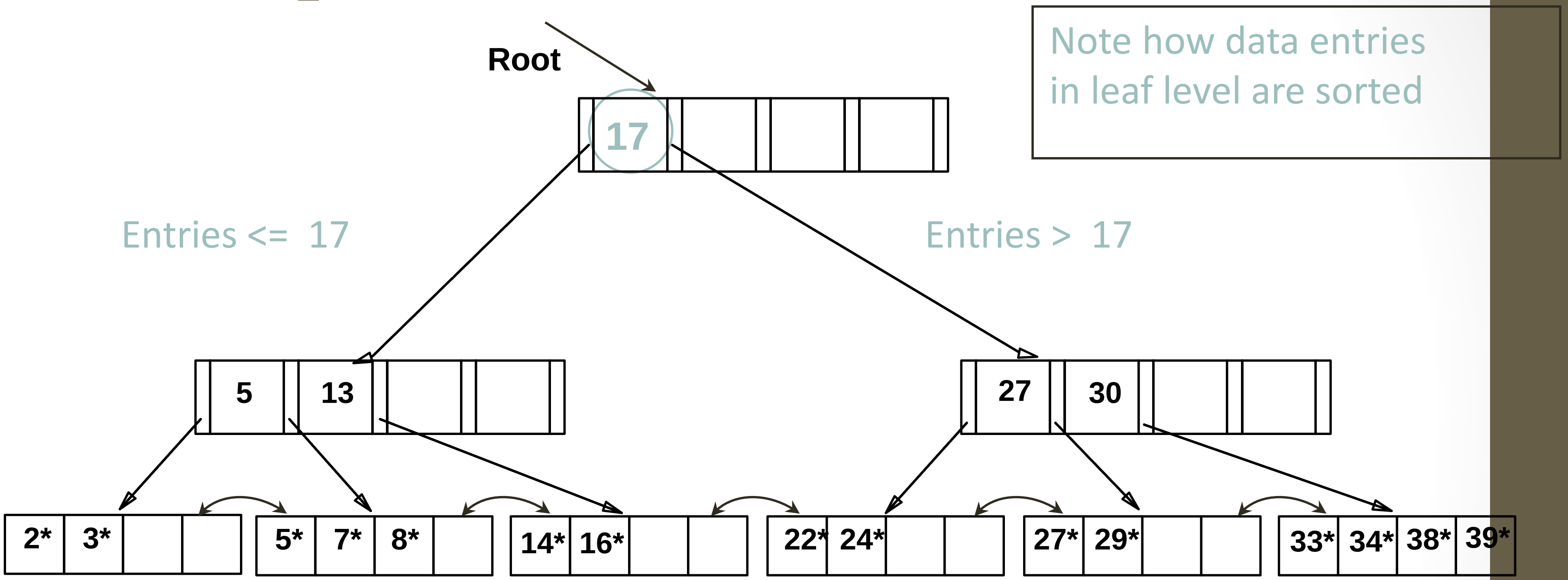
Tree Indexes



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



Note how data entries in leaf level are sorted

- 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 bucket page plus 0 or more overflow pages
  - Hashing function h: h(r) = bucket in which record r belongs
    - Function 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

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 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

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

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 data 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 (EOF)**
- **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 Log$_2$B reads to locate random record X cost of a read **Dlog$_2$B**
- **Range search**
  - Use a binary search to locate first page to satisfy criterion **Dlog$_2$B**
  - Also need a read for every other page that satisfies the criterion

    Dlog$_2$B + # pages with matching recs
- **Insert a record**
  - Search to the page for the insertion + **BD**
- **Delete a record**
  - Search to the page for the deletion + **BD**

Clustered Index 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 $\log_F 1.5B$
  - Number of disk reads (**$\log_F$ 1.5B X Time to read page D)**
- **Range search**
  - Find first page to satisfy criterion in $\log_F 1.5B$
  - Subsequent leaf nodes are read until you hit a record not satisfying the condition (**$\text{Log}_F 1.5B$ + # 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 $\log_F$. (.15B)
  - Number of disk reads **(1 + $\log_F$ .15B) X Time to read page D**
- **Range search**
  - Find first page to satisfy criterion in $\log_F$ (.15B)
  - Subsequent leaf nodes are read until you hit a record not satisfying the condition **D(Log$_F$ .15B + # pages with matching recs)**

- **Insert a record**
  - Search to the page for the Insertion for the data record in the file **2D**
  - Find insertion spot in index **DLog$_F$ .15B, do insertion D => D(3 + Log$_F$.15B)**
- **Delete a record**
  - Search to the page for the delete in the data file and in the index **DLog$_F$.15B + D**
  - Write out the modified pages **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 = RBD + .125BD Expensive
- **Equality search**
  - Find read index page **D**
  - Read data page **D** **: total 2D**
- **Range search – no help from index since hashing value**
  - Read entire heap file **BD**
- **Insert a record**
  - Read , write data record **2D + C**
  - Read, write index **2D + C** **Total cost (4D)**
- **Delete a record**
  - Search to the page for the deletion **2D**
  - Write index + data page **2D**

Understand 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
  - Which attributes are involved in selection/join conditions?
  - The type of update (INSERT/DELETE/UPDATE) and the attributes that are affected.

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?

Choice of indexes

- One approach:
  - Consider the most important queries for DB.
  - Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create the new index.
- 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.
- 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.
  - 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
```

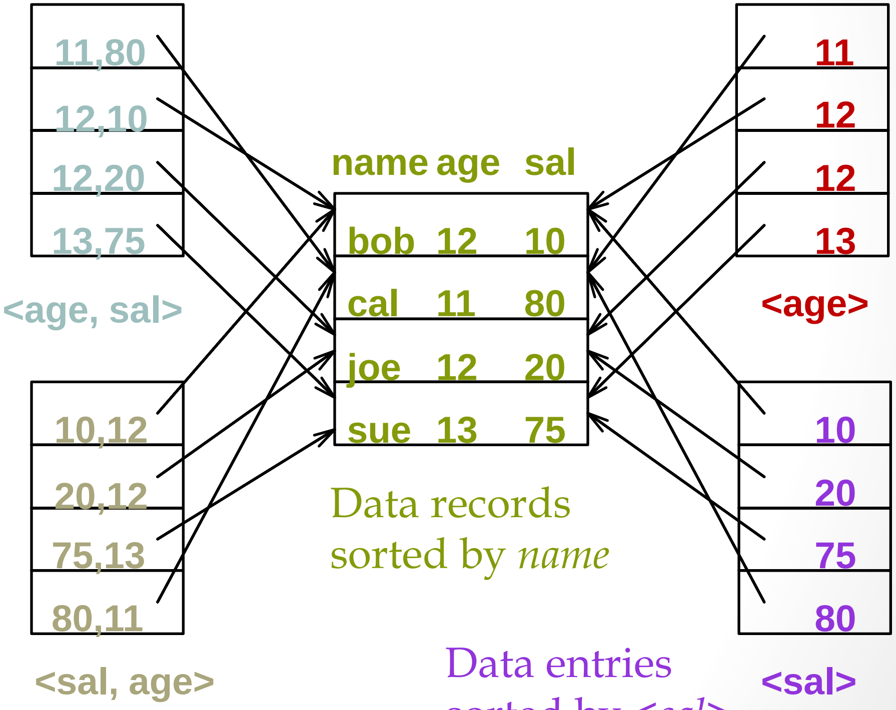
```
SELECT E.dno,  COUNT (*)
FROM Emp E
WHERE E.age>10
GROUP BY E.dno
```

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

Indexes with Composite Search Key

- *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.
  - Lexicographic order, or
  - Spatial order.

Examples of composite key indexes using lexicographic order.



name	age	sal
bob	12	10
cal	11	80
joe	12	20
sue	13	75

Data records sorted by *name*

Data entries in index sorted by *<sal,age>*

Data 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>	SELECT E.dno, COUNT(*) FROM Emp E GROUP BY E.dno
<E.dno,E.eid> Tree index	SELECT E.dno, MIN(E.sal) FROM Emp E GROUP BY E.dno
<E.dno>	SELECT E.dno, COUNT(*) FROM Emp E GROUP BY E.dno
<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 BETWEEN 3000 AND 5000

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

- 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 DB 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.