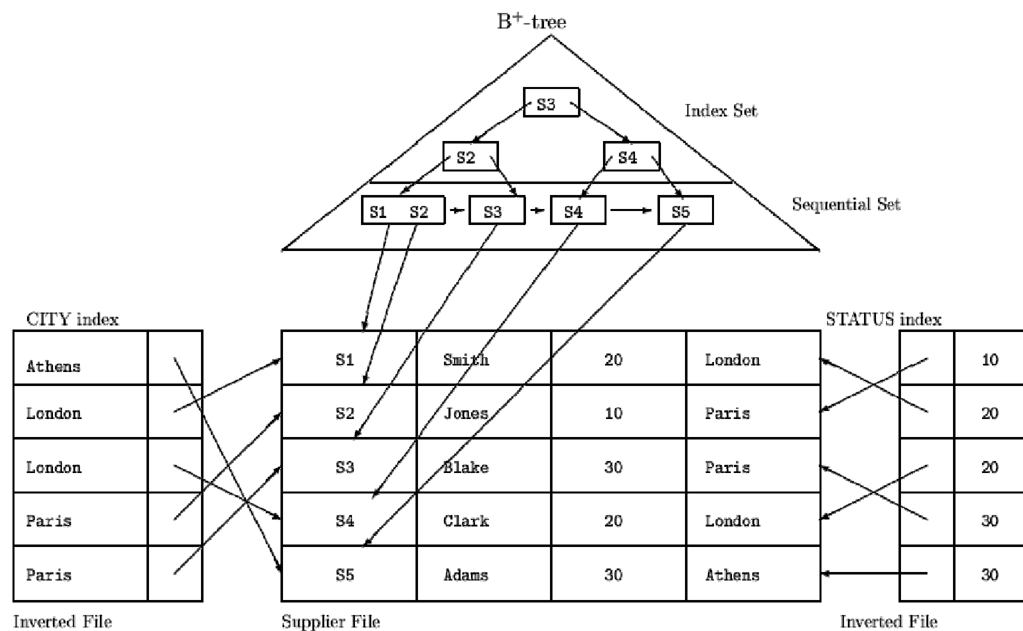
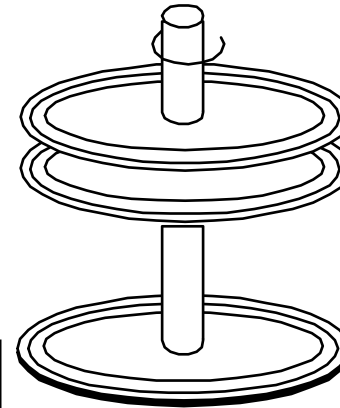
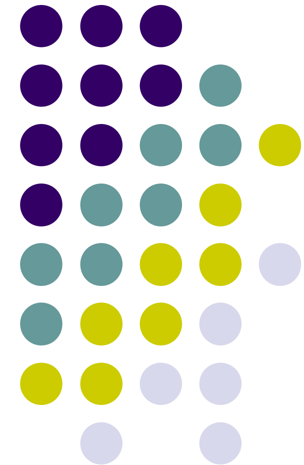
Overview of Storage and Indexing

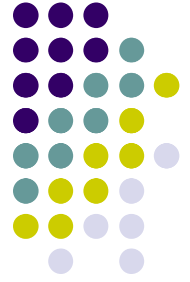
Chapter 8







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
  - In practise redundant storage is used to **avoid data-loss** (RAID Systems (disk arrays), mirror sites, regular backups (checkpoints) etc.)
- Tapes: Can only read pages in sequence
  - Cheaper than disks; used for archival storage

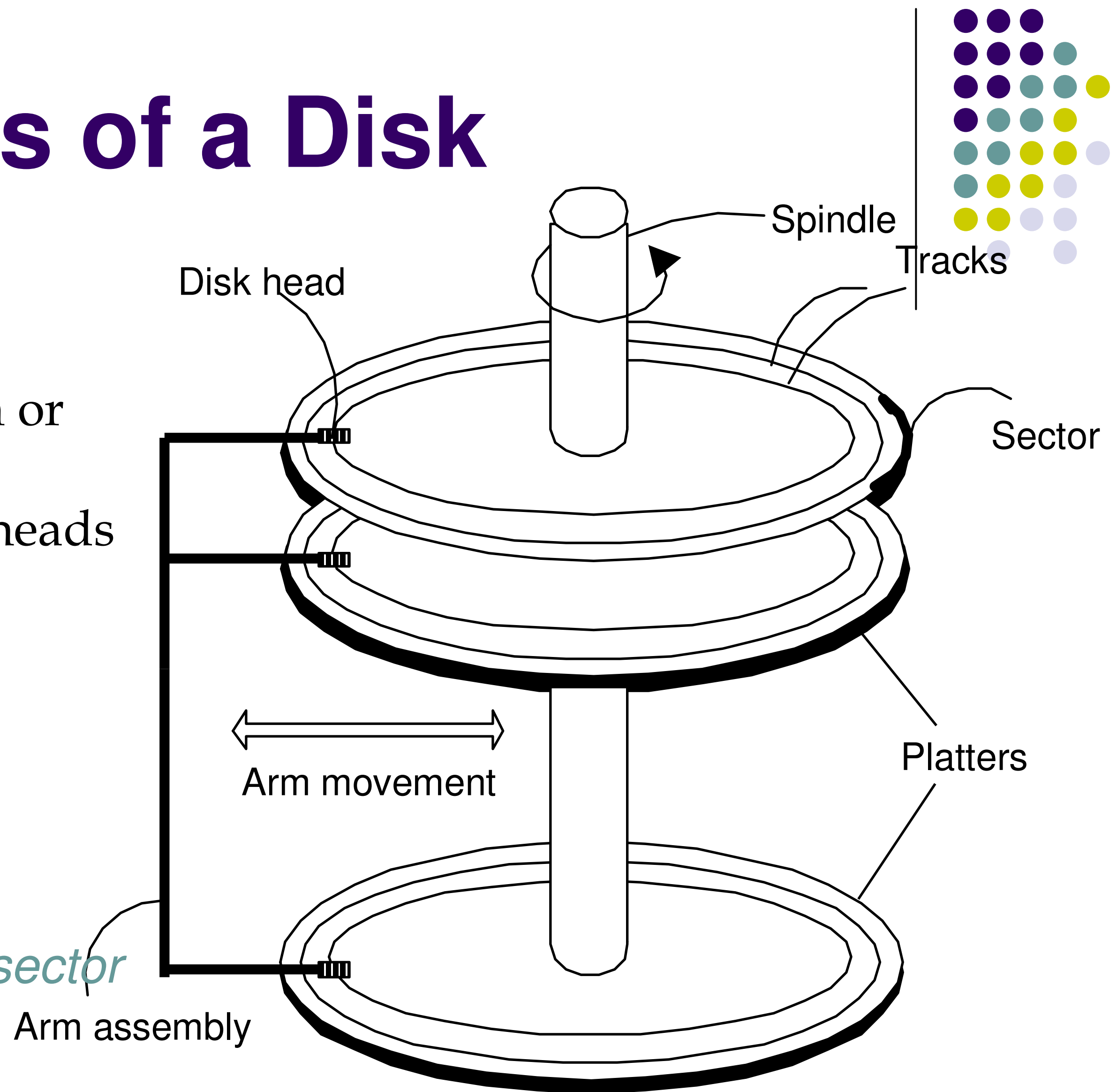
Components of a Disk

❖ Platters spin (say, 90rps).

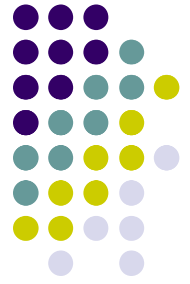
❖ Arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).

❖ Only one head reads/ writes at any one time.

❖ *Block size* is a multiple of *sector size* (which is fixed); Blocks are read at once.

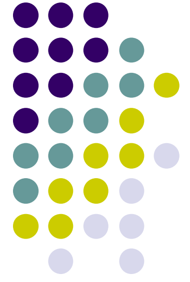


Accessing a Disk Page



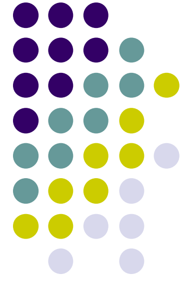
- Time to access (read/write) a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?

File Organizations,Record IDs, and Indexes



- **File organization:** Arrangment of a file of records on external storage. (Heap file, Sorted file)
- Record id (rid) is sufficient to physically locate record (typically a tuple of the database)
- Indexes are data structures that allow us to find the record ids of records with given values in index search key fields

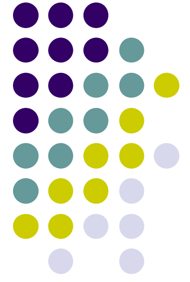
Alternative File Organizations



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

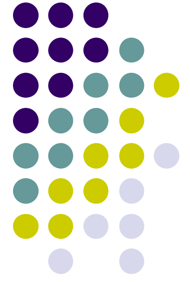
- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed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

Indexes



- 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 always the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.

B+ Tree index example (details later)



B+-tree

Index Set: S3 → S2, S4

Sequential Set: S1 S2 → S3 → S4 → S5

CITY index

Athens	
London	
London	
Paris	
Paris	

Inverted File

S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

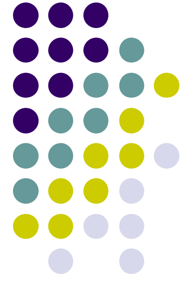
Supplier File

STATUS index

	10
	20
	20
	30
	30

Inverted File

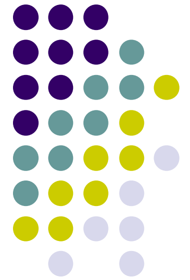
Hash index example (details later)



h: age → two-least-significant-bits(age)

30 = 1*16 + 1*8 + *4 + 1*2 +0*1 = 0110 => LSB = 10

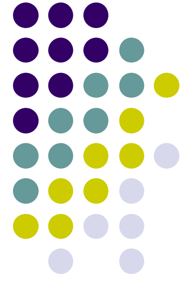
Index in SQL



There is no standard for how to create an index in SQL:1999 !

CREATE INDEX IndAgeRating ON STUDENTS
WITH STRUCTURE = BTREE
KEY = (age,gpa)

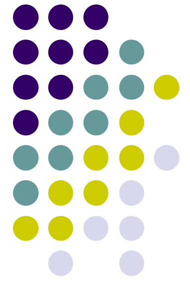
Alternatives for Data Entry k* in Index



- Three alternatives:
  - **A1** Data record with key value **k**
  - **A2** <**k**, rid of data record with search key value **k**>
  - **A3** <**k**, list of rids of data records with search key value **k**>

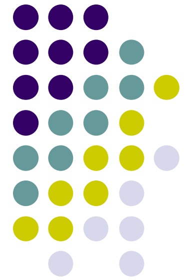
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k.**
  - Examples of indexing techniques: B+ trees, hash index
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)



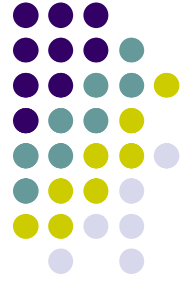
- Alternative 1: (rid+data record)
  - If this is used, index structure is a *file organization* for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage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

Alternatives for Data Entries (Contd.)



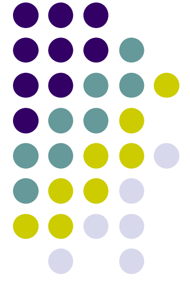
- Alternatives 2 and 3: (only rids)
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Index Classification



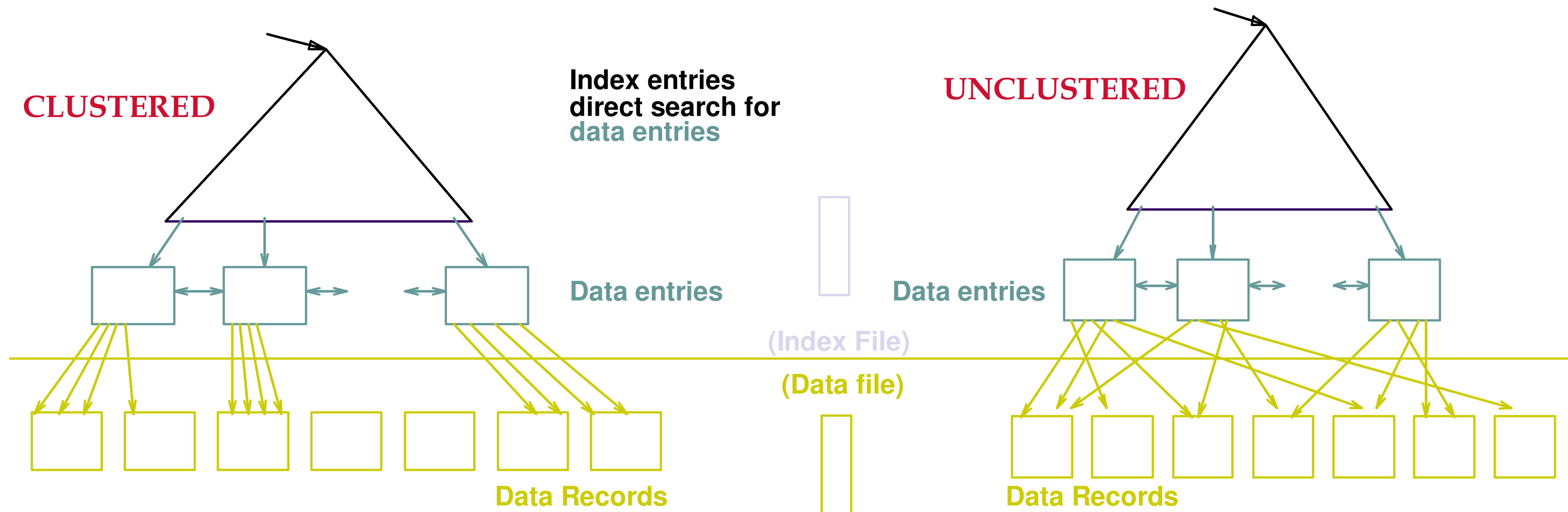
- *Primary* vs. *secondary*: If search key contains primary key, then called primary index.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to', order of data entries, then called clustered index.
  - Alternative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

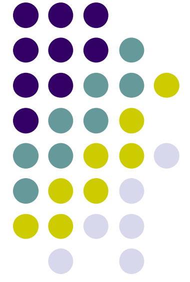


- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - *Overflow pages* may be needed for inserts.

    (Thus, order of data recs is `close to', but not identical to, the sort order.)



Hash-Based Indexes



- Index is a collection of *buckets.*

  Def.: **Bucket**:

  primary page plus zero or more *overflow* pages

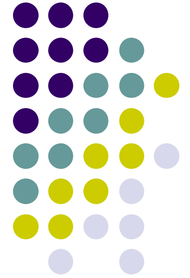
  Def::**Hashing function:**

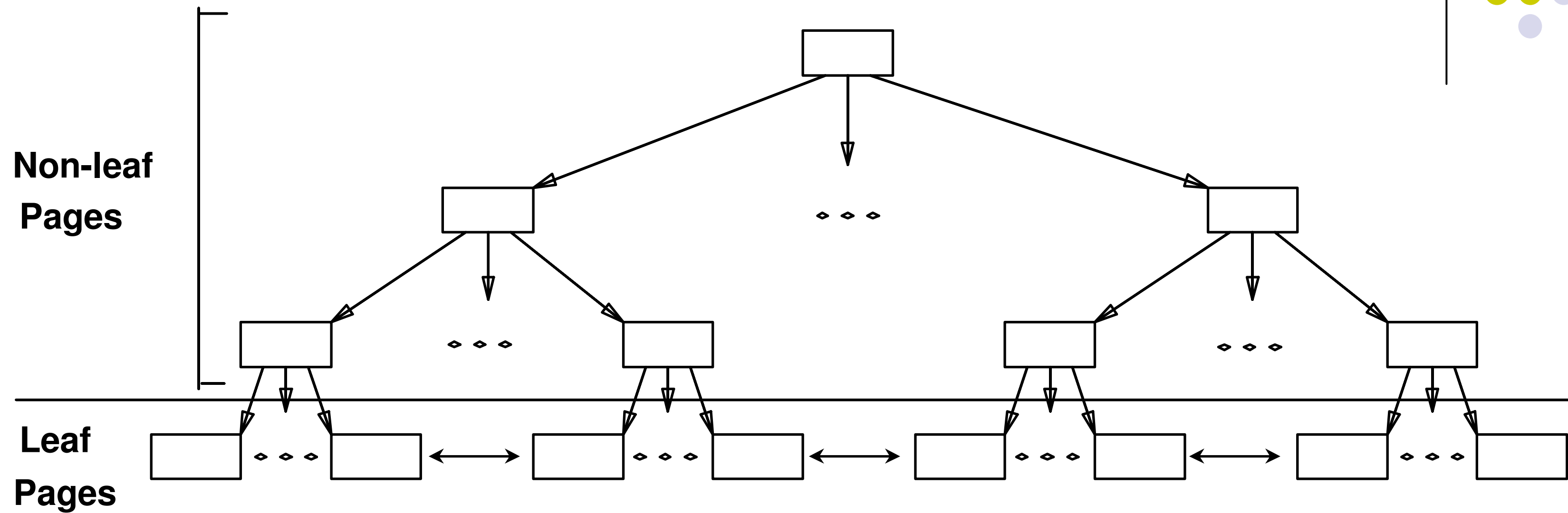
  h(*r*) = bucket in which record *r* belongs.

  h looks at the *search key* fields of *r.*

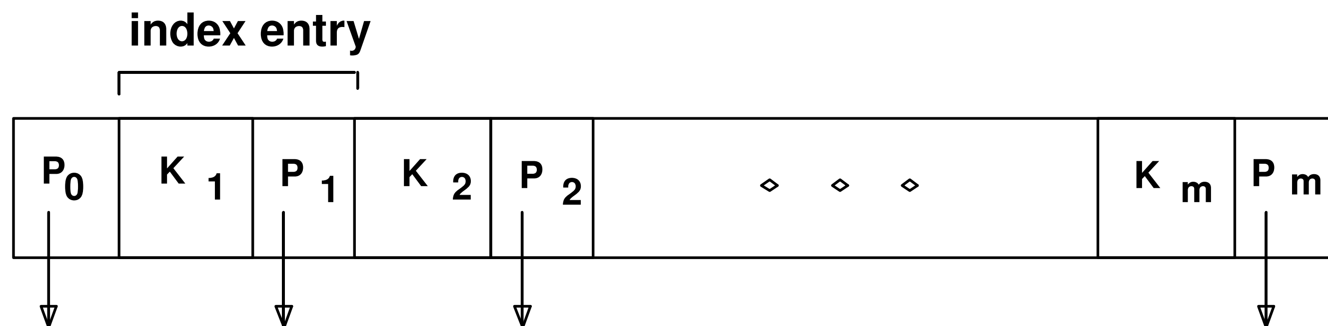
- If Alternative (1) is used, the buckets contain the data records; otherwise, they contain <key, rid> or <key, rid-list> pairs.
- Good for **equality** selection! Not good for range selection.

B+ Tree Indexes

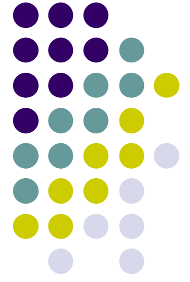


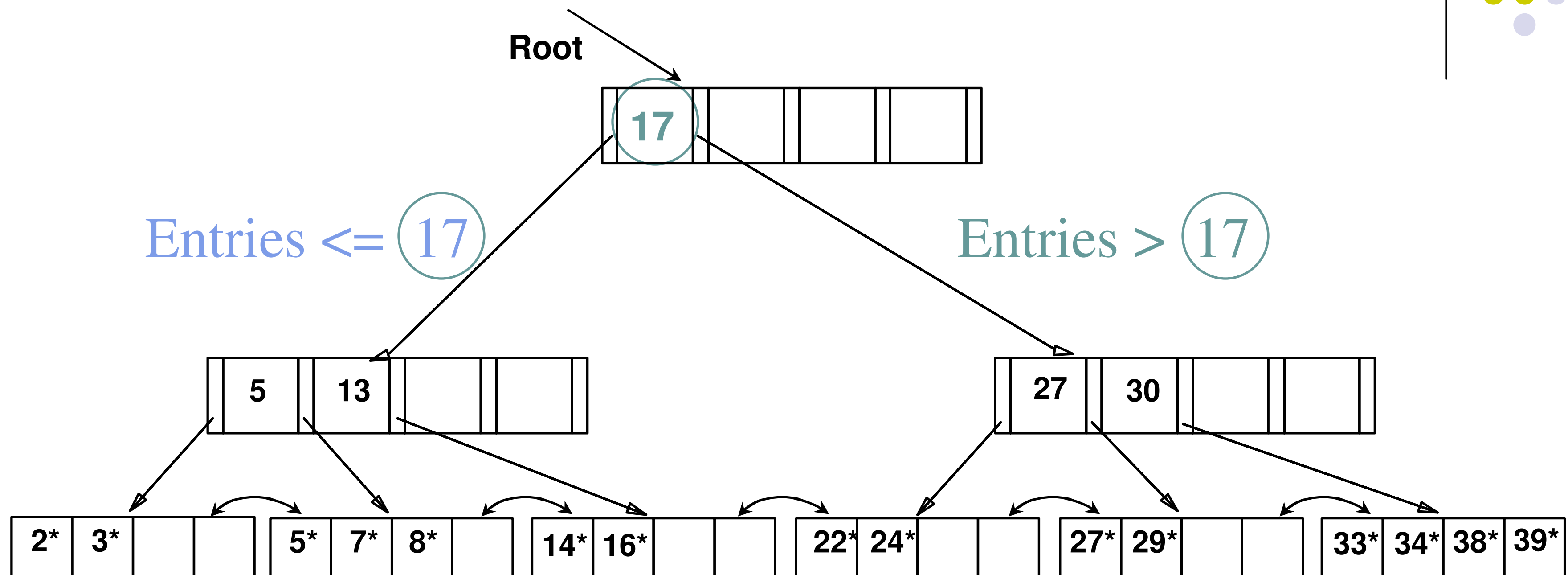


- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages contain *index entries* and direct searches:



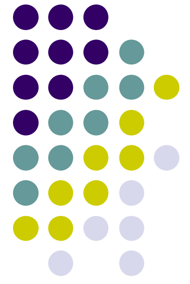
Example B+ Tree





- 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

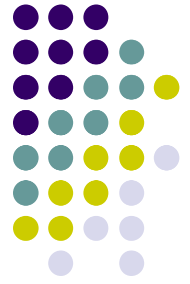


We ignore CPU costs, for simplicity:

- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write 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 simplistic assumptions.

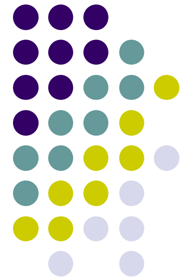
☒ *Good enough to show the overall trends!*

Comparing File Organizations



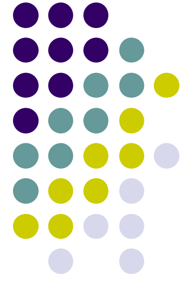
- Heap files (random order; insert at end of file)
- Sorted files, sorted on *<age, sal>*
- Clustered B+ tree file, Alternative (1), search key *<age, sal>*
- Heap file with unclustered B + tree index on search key *<age, sal>*
- Heap file with unclustered hash index on search key *<age, sal>*

Operations to Compare



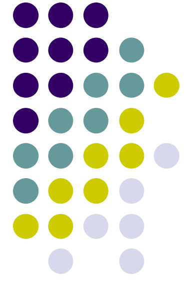
1. Scan: Fetch all records from disk
2. Equality search
3. Range selection
4. Insert a record
5. Delete a record

Heap file



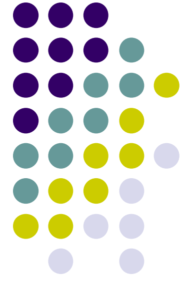
- Scan: time is BD
- Equality selection: Average time is 0.5 BD (Assumption: single match)
- Range selection: BD (Qualifying records can appear everywhere in file)
- Insert: 2D
- Delete: 0.5 BD

Sorted files



- Scan: BD
- Equality selection: $Dlog_2B$
- Range Selection: $Dlog_2B$ + #Pages with qualified entries
- Insert: $Dlog_2B + 2(0.5\ BD)$
- Delete: (as insert) $Dlog_2B + 2(0.5\ BD)$

Clustered B+ tree

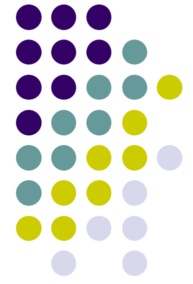


- Scan: 1.5BD (Assumption: 67% page fill)

  → use linked leaf list
- Search with equality selection: D $\log_F$ 1.5B
- Search with range selection: D$\log_F$1.5B+#matching records (linked list can be used)
- Insert: (leaf has enough space):

  D $\log_F$ 1.5B + D
- Delete: like insert

Unclustered B+ tree index

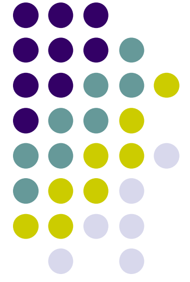


Reading all data entries

Page reads for index informatio

- Scan: BDR + 0.15 BD
  - Assumption: storage(data entry) = 0.1 storage(record), Factor 1.5: (67% page
  - fill)
- Search equality selection: D $\log_F 0.15$ B + D
- Search range selection: D $\log_F 0.15$ B + #matching records D
- Insert: 2D (read and write) + D $\log_F 0.15$ B
- Delete: 2D + D $\log_F 0.15$ B

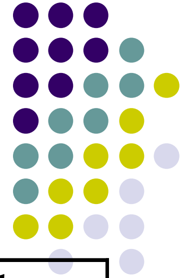
Unclustered Hash index



- Static hashing with no overflow chains
- Hash page has 80% occupancy
- 10(0.8R) = 8R entries fit on a page
- Scan: BRD, results are unordered

  ➔Prohibitivly expensive, no-one uses hash index for scan !
- Search with equality selection: H+D+D (Read bucket page and read page with record)
- Search with range selection: BD
- Insert: H+2D+2D (locate and update bucket page + update page with record)
- Delete: H+2D+2D

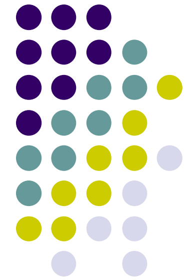
Cost of Operations



	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap					
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

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

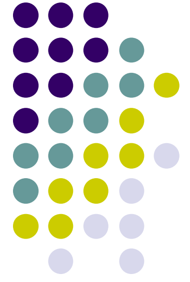
Cost of Operations



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

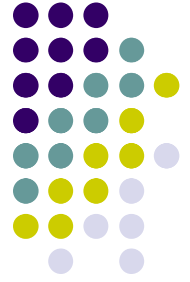
✉ *Several assumptions underlie these (rough) estimates!*

Summary Cost Analysis



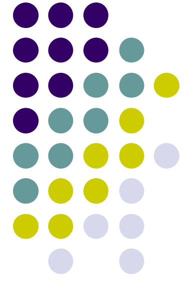
- Heap files: Fast insert, low extra storage, slow search and scan, slow delete
- Sorted files: Fast search, low extra storage, expensive insert and delete
- B+ trees (clustered): Very fast equality search and range selection, insert and delete, fast range selection, memory overhead (ca. 1.5), supports composite keys
- Hash: Very fast equality search, insert and delete, slow range selection

Understanding the Workload



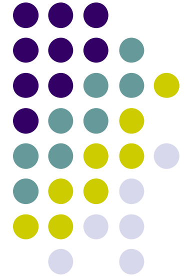
- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join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

Choice of Indexes



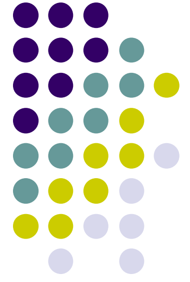
- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree?

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.

Examples of Clustered Indexes



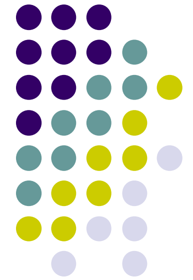
- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?
- Consider the GROUP BY query.
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!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

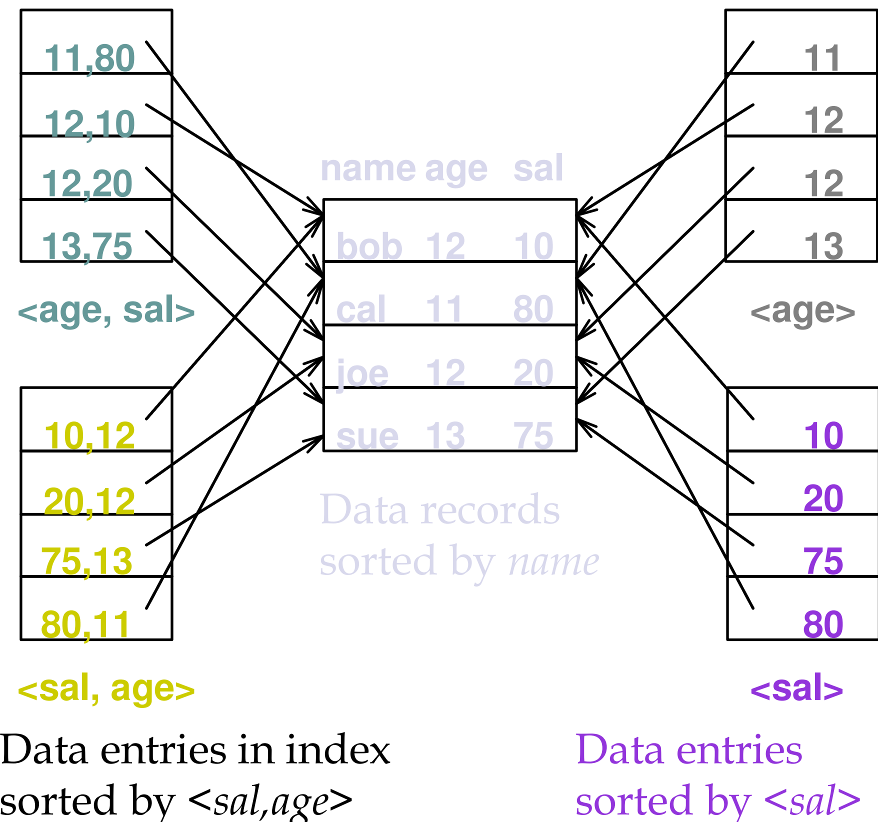
```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

Indexes with Composite Search 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.
  - Lexicographic order, or
  - Spatial order.

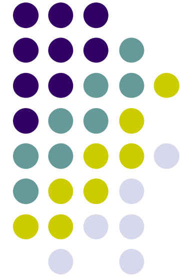
Examples of composite key indexes using lexicographic order.



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

Data entries sorted by <*sal*>

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.

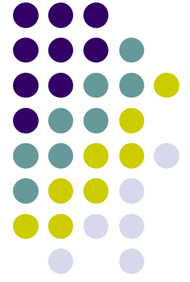
<*E.dno,E.eid*> *Tree index!*

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

<*E.dno*>

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

Index Selection Guidelines



- 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.
    - For index-only strategies, clustering is not important!
- 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.