

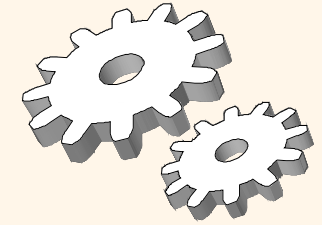
# *Overview of Storage and Indexing*

## Chapter 8

*One of the great dividends of investing in an RDBMS is that you don't have to think too much about the computer's inner life.*

*You're the programmer and say what kinds of data you want. The computer's job is to fetch it and you don't really care how.*

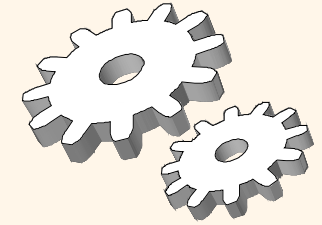
*Philip Greenspun in "SQL for Web Nerds"*



# *Data on External Storage*

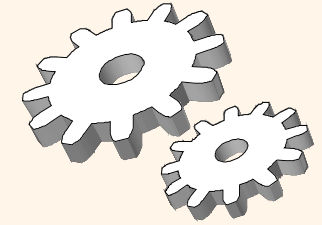
- ❖ Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order
- ❖ Tapes: Can only read pages in sequence
  - Cheaper than disks; used for archival storage
- ❖ File organization: Method of arranging a file of records on external storage.
  - **Record id (rid)** is sufficient to physically locate record
  - **Indexes** are data structures that allow us to find the record ids of records with given values in **index search key** fields
- ❖ Architecture: **Buffer manager** stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

# Alternative File Organizations



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

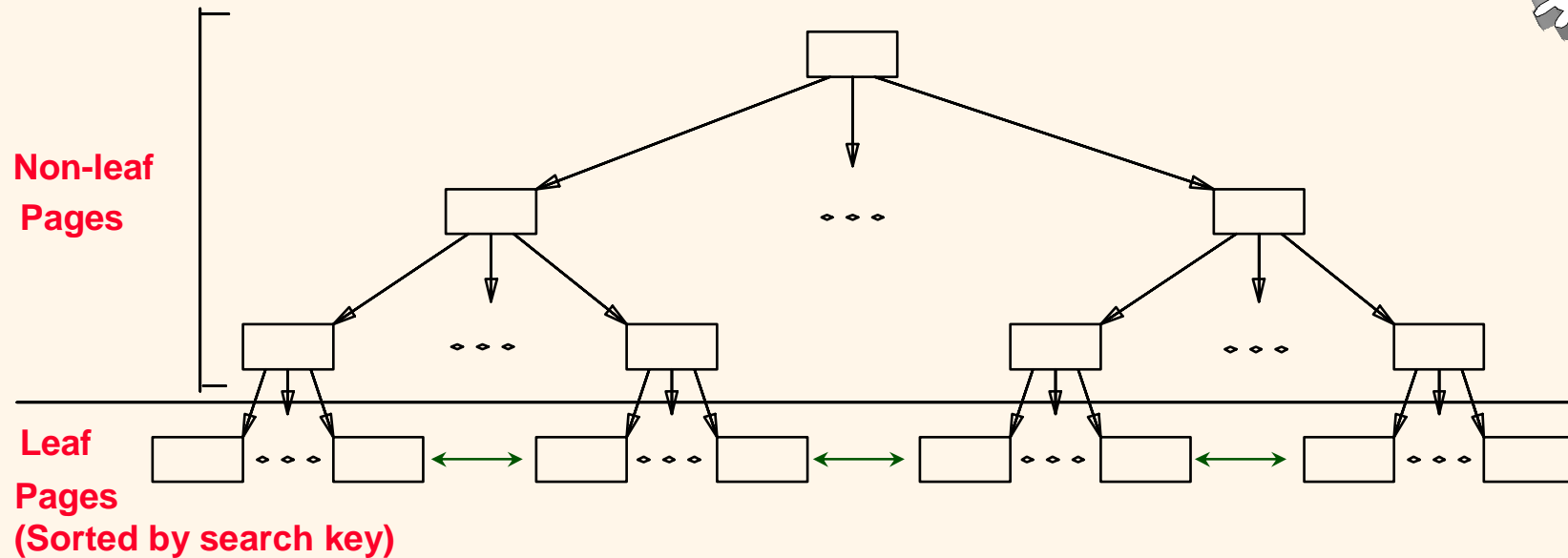
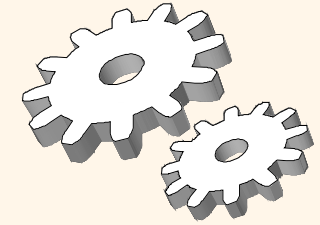
- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range` of records is needed.
- Indexes: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed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



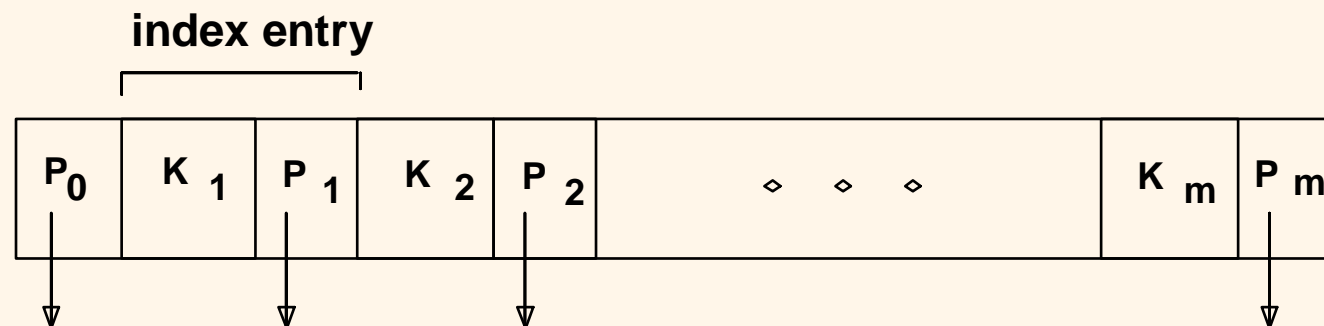
# Indexes

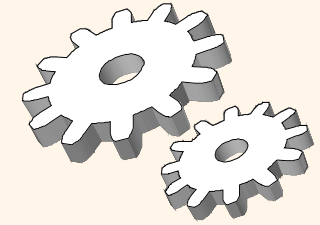
- ❖ An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is **not** the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- ❖ An index contains a collection of *data entries*, and supports efficient retrieval of all data entries  **$k^*$**  with a given key value  **$k$** .
  - Given data entry  $k^*$ , we can find record with key  $k$  in at most one disk I/O. (Details soon ...)

# B+ Tree Indexes

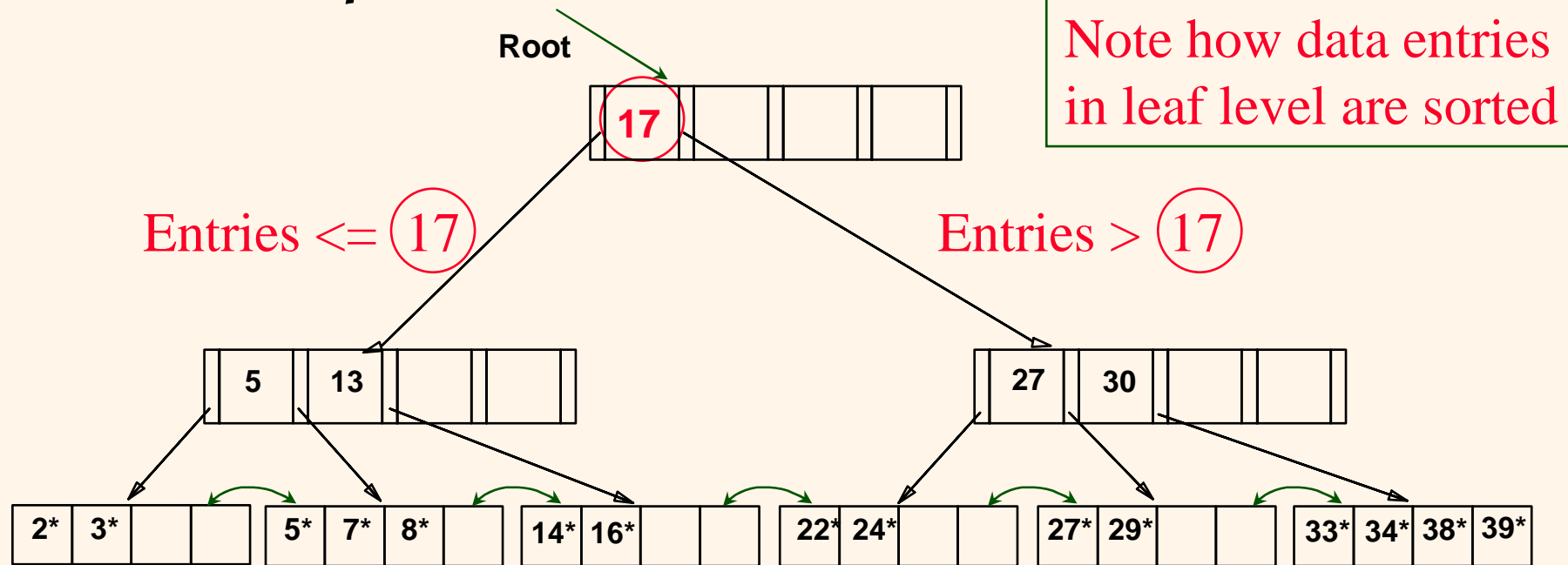


- ❖ Leaf pages contain *data entries*, and are chained (prev & next)
- ❖ Non-leaf pages have *index entries*; only used to direct searches:

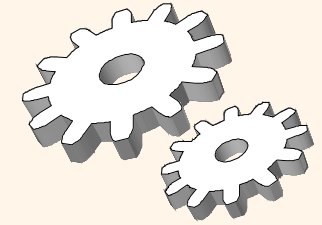




# Example B+ Tree

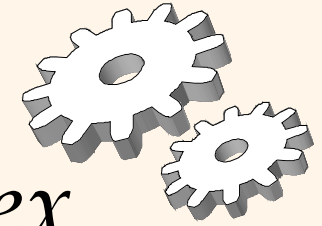


- ❖ Find 28\*? 29\*? All  $> 15^*$  and  $< 30^*$
- ❖ Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree



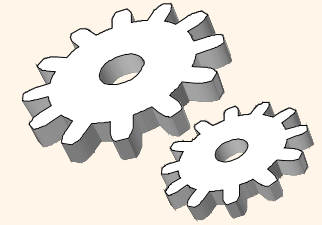
# Hash-Based Indexes

- ❖ 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  $r$ .
  - *No need for “index entries” in this scheme.*



## *Alternatives for Data Entry $k^*$ in Index*

- ❖ In a data entry  $k^*$  we can store:
  - Data record with key value  $k$ , or
  - $\langle k, \text{rid of data record with search key value } k \rangle$ , or
  - $\langle k, \text{list of rids of data records with search key } k \rangle$
- ❖ Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value  $k$ .
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries

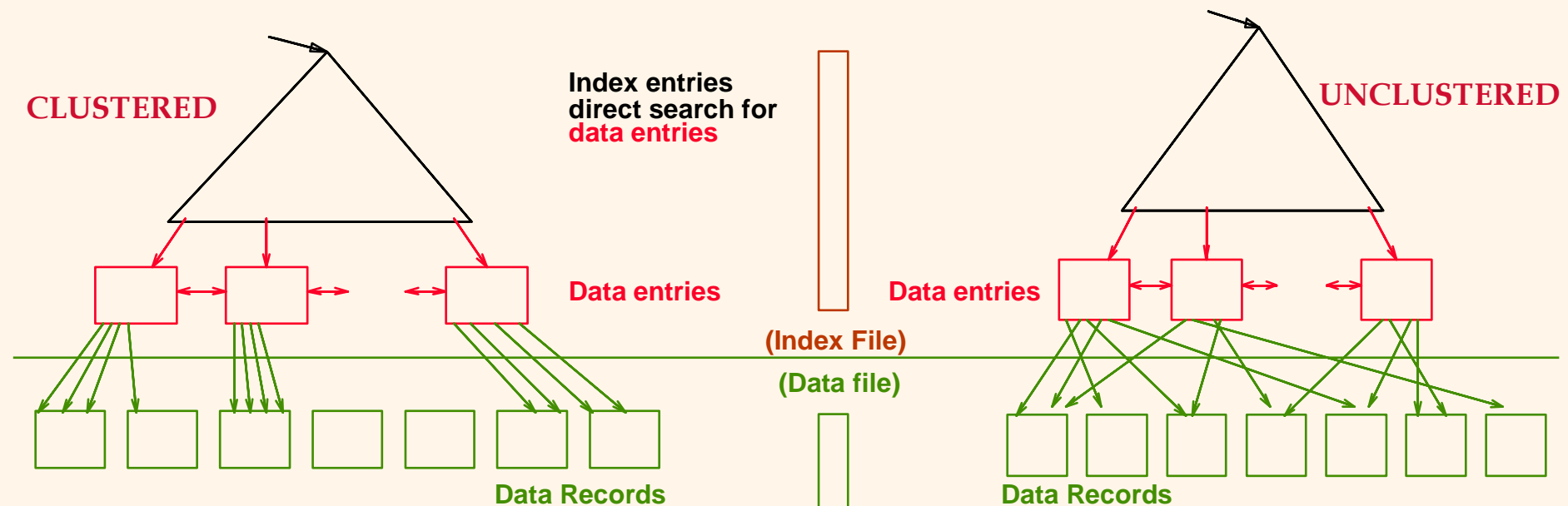


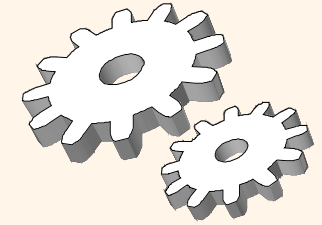
# *Index Classification*

- ❖ *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.
  - 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.)

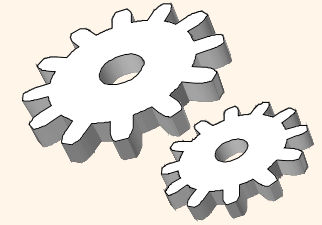




## *Choice of Indexes (Contd.)*

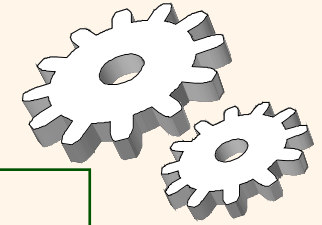
- ❖ **One approach:** Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates **query evaluation 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



- ❖ 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.

# 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
```

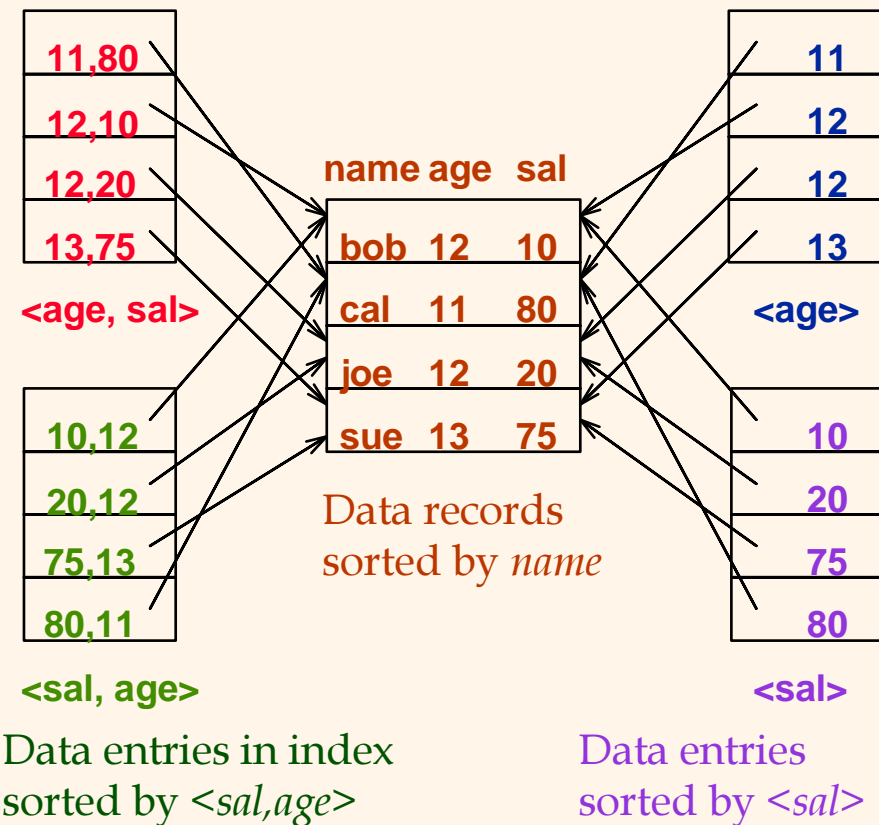
```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age>10
GROUP BY E.dno
```

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

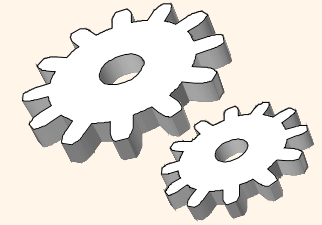
# Indexes with Composite Search Keys

- ❖ **Composite Search Keys:** Search on a combination of fields.
  - **Equality query:** Every field value is equal to a constant value. E.g. wrt  $\langle \text{sal}, \text{age} \rangle$  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.

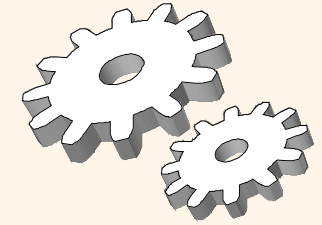


# Composite Search Keys



- ❖ To retrieve Emp records with  $age=30$  AND  $sal=4000$ , an index on  $\langle age, sal \rangle$  would be better than an index on  $age$  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  $\langle age, sal \rangle$  or  $\langle sal, age \rangle$  is best.
- ❖ If condition is:  $age=30$  AND  $3000 < sal < 5000$ :
  - Clustered  $\langle age, sal \rangle$  index much better than  $\langle sal, age \rangle$  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.

$\langle E.dno \rangle$

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

$\langle E.dno, E.sal \rangle$   
*Tree index!*

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

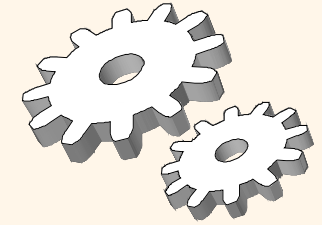
$\langle E.age, E.sal \rangle$

or

$\langle E.sal, E.age \rangle$

*Tree index!*

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
E.sal BETWEEN 3000 AND 5000
```



## *Index-Only Plans (Contd.)*

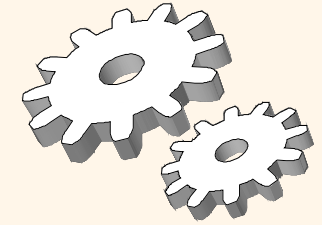
❖ Index-only plans are possible if the key is  $\langle \text{dno}, \text{age} \rangle$  or we have a tree index with key  $\langle \text{age}, \text{dno} \rangle$

- Which is better?
- What if we consider the second query?

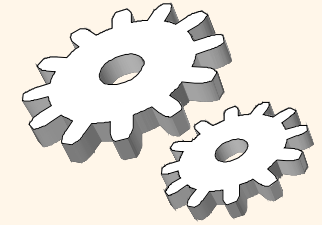
```
SELECT E.dno, COUNT (*)  
FROM Emp E  
WHERE E.age=30  
GROUP BY E.dno
```

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

# Summary



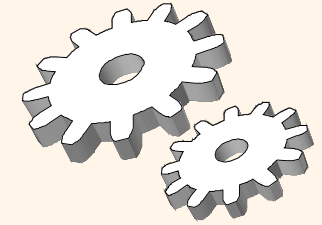
- ❖ Many alternative file organizations exist, each appropriate in some situation.
- ❖ If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good 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.



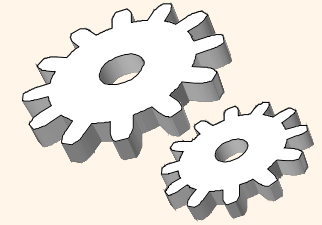
## *Summary (Contd.)*

- ❖ 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, and dense vs. sparse. Differences have important consequences for utility/performance.

# Summary (Contd.)



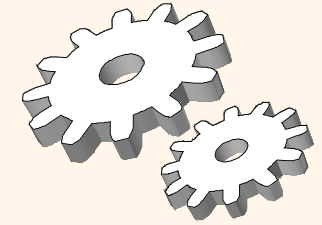
- ❖ Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.



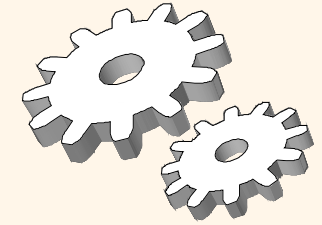
# *Overview of Query Evaluation*

## Chapter 12

# Overview of Query Evaluation



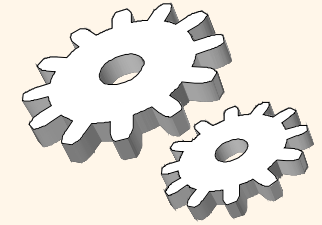
- ❖ Plan: Tree of Relational Algebra operations, with choice of algorithms for each operation.
  - Each operator typically implemented using a `pull` interface: when an operator is `pulled` for the next output tuples, it `pulls` on its inputs and computes them.
- ❖ Two main issues in query optimization:
  - For a given query, **what plans are considered?**
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the **cost of a plan estimated?**
- ❖ **Ideally**: Want to find best plan. **Practically**: Avoid worst plans!



## *Some Common Techniques*

- ❖ Algorithms for evaluating relational operators use some simple ideas extensively:
  - **Indexing:** Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - **Iteration:** Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - **Partitioning:** By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

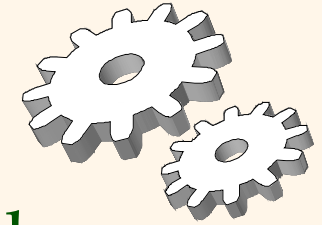
*\* Watch for these techniques as we discuss query evaluation!*



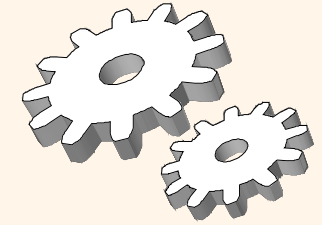
# Statistics and Catalogs

- ❖ Need information about the relations and indexes involved. *Catalogs* typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- ❖ Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- ❖ More detailed information (e.g., histograms of the values in some field) are sometimes stored.

# Access Paths



- ❖ An access path is a method of retrieving tuples:
  - **File scan**, or **index** that **matches** a selection (in the query)
- ❖ A tree index matches (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - E.g., Tree index on  $\langle a, b, c \rangle$  **matches** the selection  $a=5$  **AND**  $b=3$ , and  $a=5$  **AND**  $b>6$ , but not  $b=3$ .
- ❖ A hash index matches (a conjunction of) terms that has a term *attribute = value* for every attribute in the search key of the index.
  - E.g., Hash index on  $\langle a, b, c \rangle$  **matches**  $a=5$  **AND**  $b=3$  **AND**  $c=5$ ; but it does not match  $b=3$ , or  $a=5$  **AND**  $b=3$ , or  $a>5$  **AND**  $b=3$  **AND**  $c=5$ .

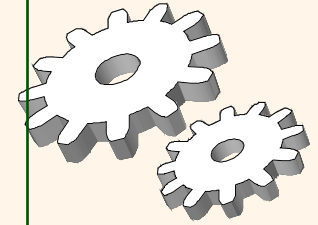


# One Approach to Selections

- ❖ Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don't **match** the index:
  - *Most selective access path*: An index or file scan that we estimate will require the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
  - Consider *day < 8/9/94 AND bid=5 AND sid=3*. A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple. Similarly, a hash index on  $\langle bid, sid \rangle$  could be used; *day < 8/9/94* must then be checked.

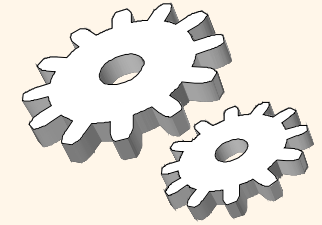
# Projection

```
SELECT  DISTINCT  
        R.sid, R.bid  
FROM    Reserves R
```



- ❖ The expensive part is removing duplicates.
  - SQL systems don't remove duplicates unless the keyword `DISTINCT` is specified in a query.
- ❖ Sorting Approach: Sort on `<sid, bid>` and remove duplicates. (Can optimize this by dropping unwanted information while sorting.)
- ❖ Hashing Approach: Hash on `<sid, bid>` to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.
- ❖ If there is an index with both `R.sid` and `R.bid` in the search key, may be cheaper to sort data entries!

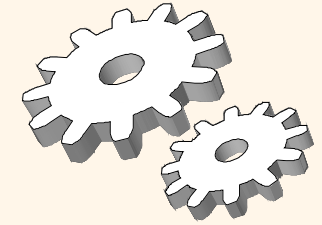
# Join: Index Nested Loops



```
foreach tuple r in R do
    foreach tuple s in S where  $r_i == s_j$  do
        add  $\langle r, s \rangle$  to result
```

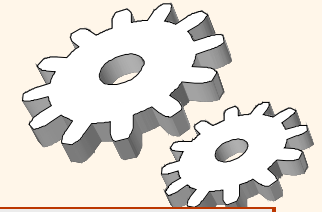
- ❖ If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost:  $M + (M * p_R) * \text{cost of finding matching S tuples}$
  - $M = \# \text{pages of R}$ ,  $p_R = \# \text{R tuples per page}$
- ❖ For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

# Join: Sort-Merge ( $R \bowtie_{i=j} S$ )



- ❖ Sort R and S on the join column, then scan them to do a “merge” (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple  $\geq$  current S tuple, then advance scan of S until current S-tuple  $\geq$  current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in  $R_i$  (*current R group*) and all S tuples with same value in  $S_j$  (*current S group*) match; output  $\langle r, s \rangle$  for all pairs of such tuples.
  - Then resume scanning R and S.
- ❖ R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

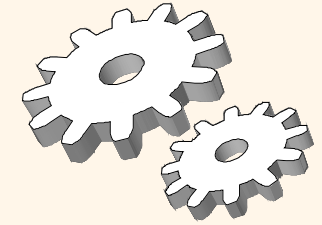
# Example of Sort-Merge Join



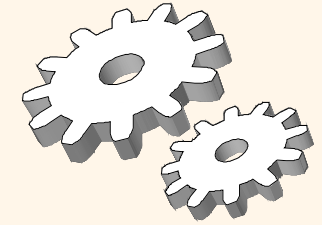
| <u>sid</u> | sname  | rating | age  | <u>sid</u> | <u>bid</u> | <u>day</u> | rname  |
|------------|--------|--------|------|------------|------------|------------|--------|
| 22         | dustin | 7      | 45.0 | 28         | 103        | 12/4/96    | guppy  |
| 28         | yuppy  | 9      | 35.0 | 28         | 103        | 11/3/96    | yuppy  |
| 31         | lubber | 8      | 55.5 | 31         | 101        | 10/10/96   | dustin |
| 44         | guppy  | 5      | 35.0 | 31         | 102        | 10/12/96   | lubber |
| 58         | rusty  | 10     | 35.0 | 31         | 101        | 10/11/96   | lubber |
|            |        |        |      | 58         | 103        | 11/12/96   | dustin |

- ❖ **Cost:  $M \log M + N \log N + (M+N)$** 
  - The cost of scanning,  $M+N$ , could be  $M*N$  (very unlikely!)
- ❖ With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

# Highlights of the Optimizer

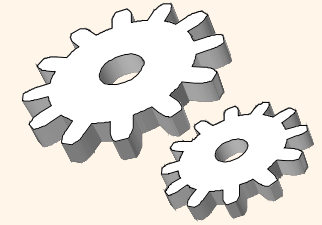


- ❖ **Cost estimation:** Approximate art at best.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- ❖ **Plan Space:** Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be pipelined into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.



# Cost Estimation

- ❖ For each plan considered, must estimate cost:
  - Must *estimate cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also *estimate size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.



# Schema for Examples

Sailors (sid: integer, sname: string, rating: integer, age: real)

Reserves (sid: integer, bid: integer, day: dates, rname: string)

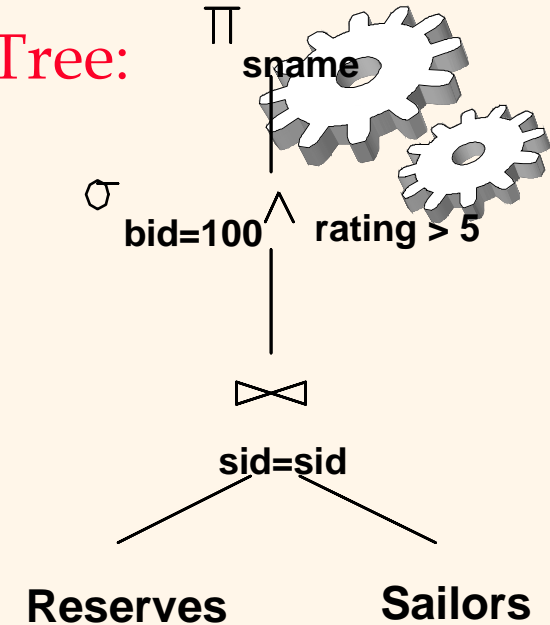
- ❖ Similar to old schema; *rname* added for variations.
- ❖ Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- ❖ Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

# Motivating Example

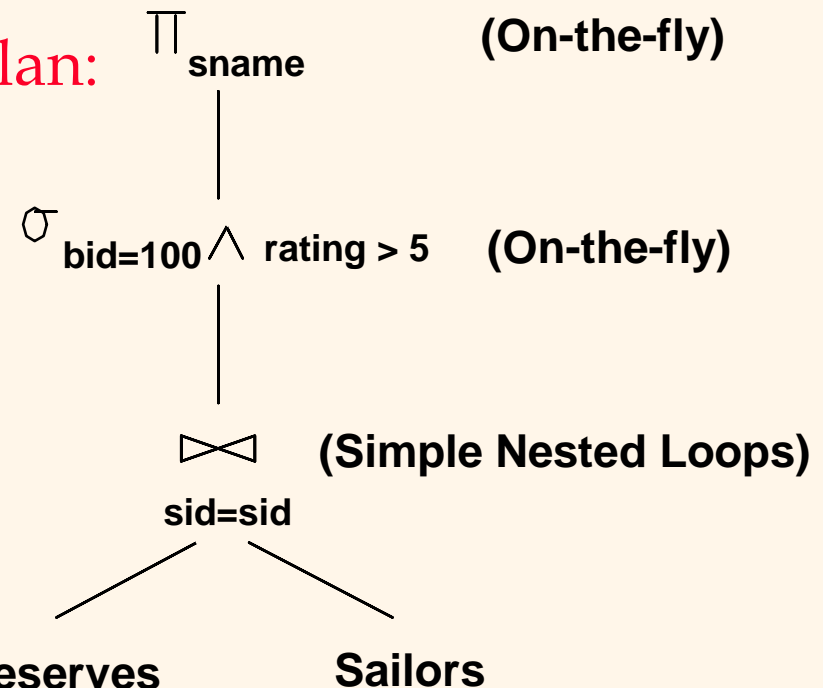
```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
      R.bid=100 AND S.rating>5
```

- ❖ **Cost: 500+500\*1000 I/Os**
- ❖ By no means the worst plan!
- ❖ Misses several opportunities: selections could have been 'pushed' earlier, no use is made of any available indexes, etc.
- ❖ *Goal of optimization:* To find more efficient plans that compute the same answer.

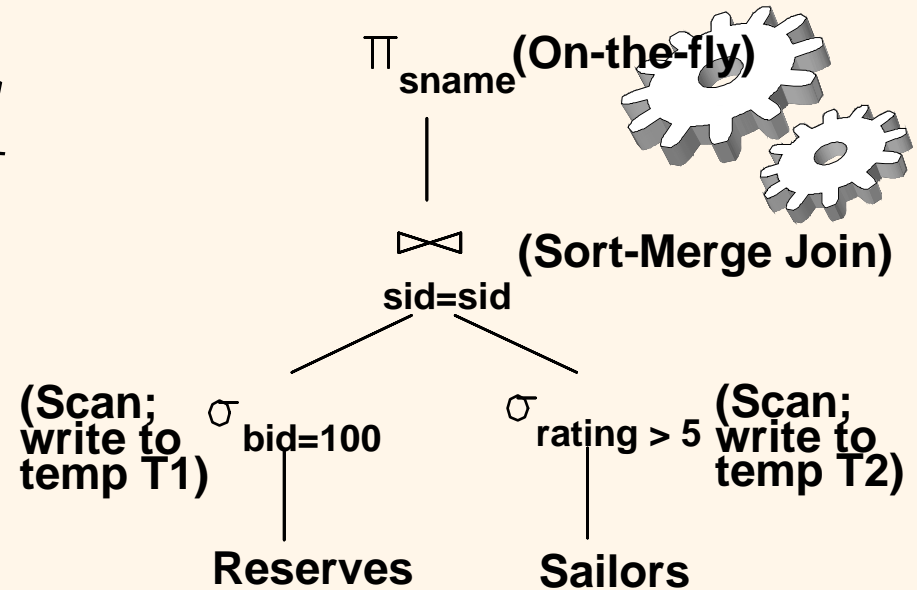
RA Tree:



Plan:



# Alternative Plans 1 (No Indexes)



- ❖ **Main difference: push selects.**
- ❖ **With 5 buffers, cost of plan:**
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort T1 ( $2 \times 2 \times 10$ ), sort T2 ( $2 \times 3 \times 250$ ), merge (10+250)
  - **Total: 3560 page I/Os.**
- ❖ **If we used BNL join, join cost =  $10 + 4 \times 250$ , total cost = 2770.**
- ❖ **If we 'push' projections, T1 has only *sid*, T2 only *sid* and *sname*:**
  - T1 fits in 3 pages, cost of BNL drops to under 250 pages, **total < 2000.**

# Alternative Plans 2 With Indexes

- ❖ With clustered index on *bid* of Reserves, we get  $100,000/100 = 1000$  tuples on  $1000/100 = 10$  pages.
- ❖ INL with pipelining (outer is not materialized).

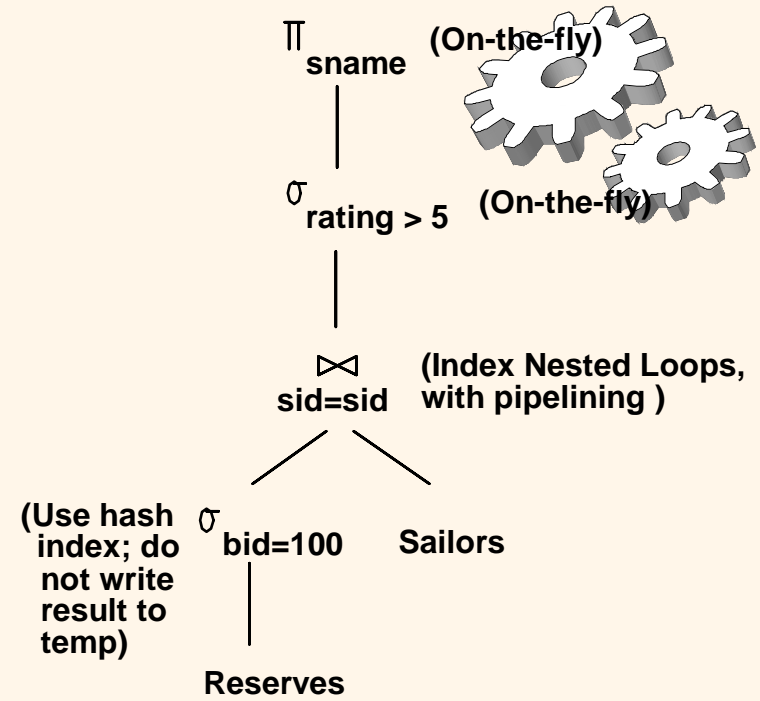
-Projecting out unnecessary fields from outer doesn't help.

- ✓ Join column *sid* is a key for Sailors.

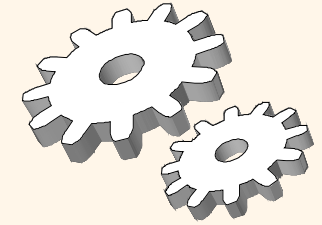
-At most one matching tuple, unclustered index on *sid* OK.

- ✓ Decision not to push *rating*>5 before the join is based on availability of *sid* index on Sailors.

- ✓ **Cost:** Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple ( $1000 \cdot 1.2$ ); total **1210 I/Os**.



# Summary



- ❖ There are several alternative evaluation algorithms for each relational operator.
- ❖ A query is evaluated by converting it to a tree of operators and evaluating the operators in the tree.
- ❖ Must understand query optimization in order to fully understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- ❖ Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.