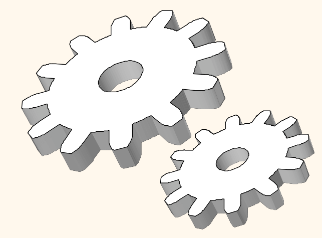
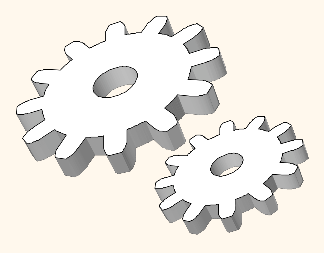


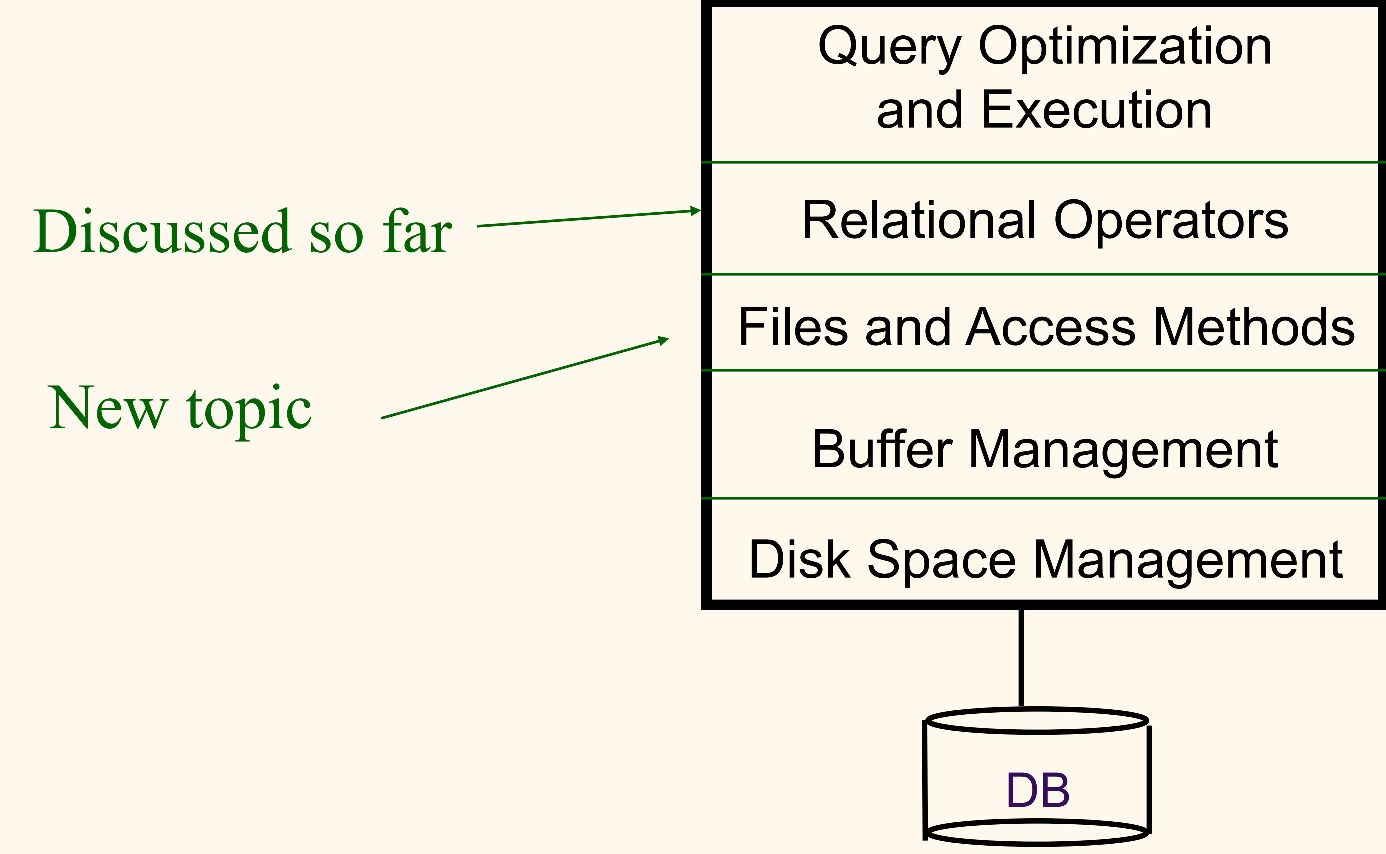
Overview of Storage and Indexing

Chapter 8

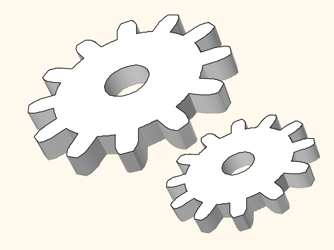
"How index-learning turns no student pale
Yet holds the eel of science by the tail."
-- Alexander Pope (1688-1744)

System Issues: How to Build a DBMS



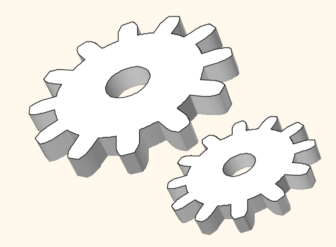


Data on External Storage



- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order
- Tapes: Can read pages only 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

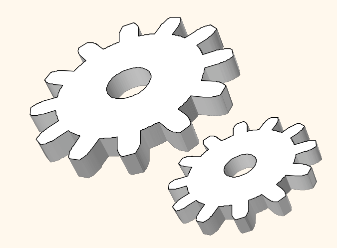
- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation (e.g., age or colour).
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in 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**.
- Example of Index: Essentials of Game Theory

*Alternatives for Data Entry **k*** in Index*

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

Alternatives for Data Entries (Contd.)

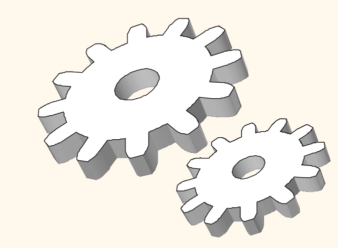
- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential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



Example of Alternative 1

Loca-tion	shape	colour	holes
1	round	Red	2
2	square	Red	4
3	rectangle	Red	8
4	round	blue	2
5	square	blue	4
6	rectangle	blue	8

6 data entries, sorted by colour

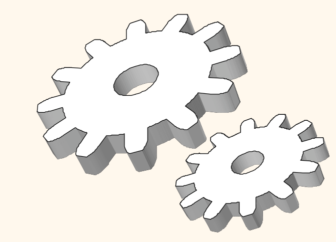


Example of Alternative 2

Loca-tion	colour
1	Red
2	Red
3	Red
4	blue
5	blue
6	blue

6 data entries, sorted by colour

Example of Alternative 3



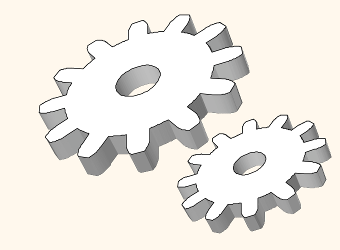
Loca-tions	colour
1, 2, 3	Red
4,5,6	Blue

2 data entries,
variable lenth

Alternatives for Data Entries (Contd.)

- Alternatives 2 and 3:
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (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.

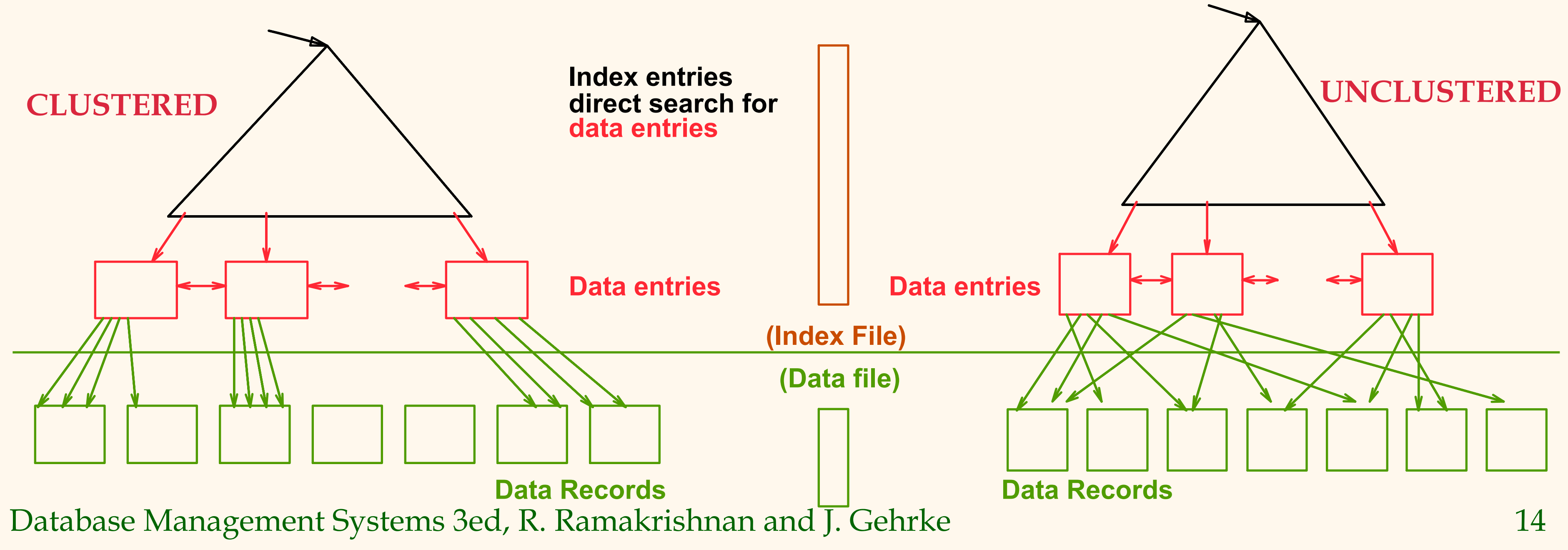
Index Classification



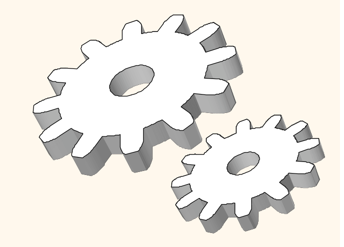
- *Primary* vs. *secondary*: If search key contains primary key, then called primary index.
  - *Unique* index: Search key uniquely identifies record.
- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to', order of data entries, then called clustered index.
  - Alternative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

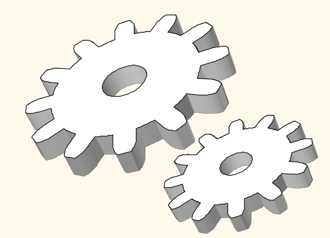


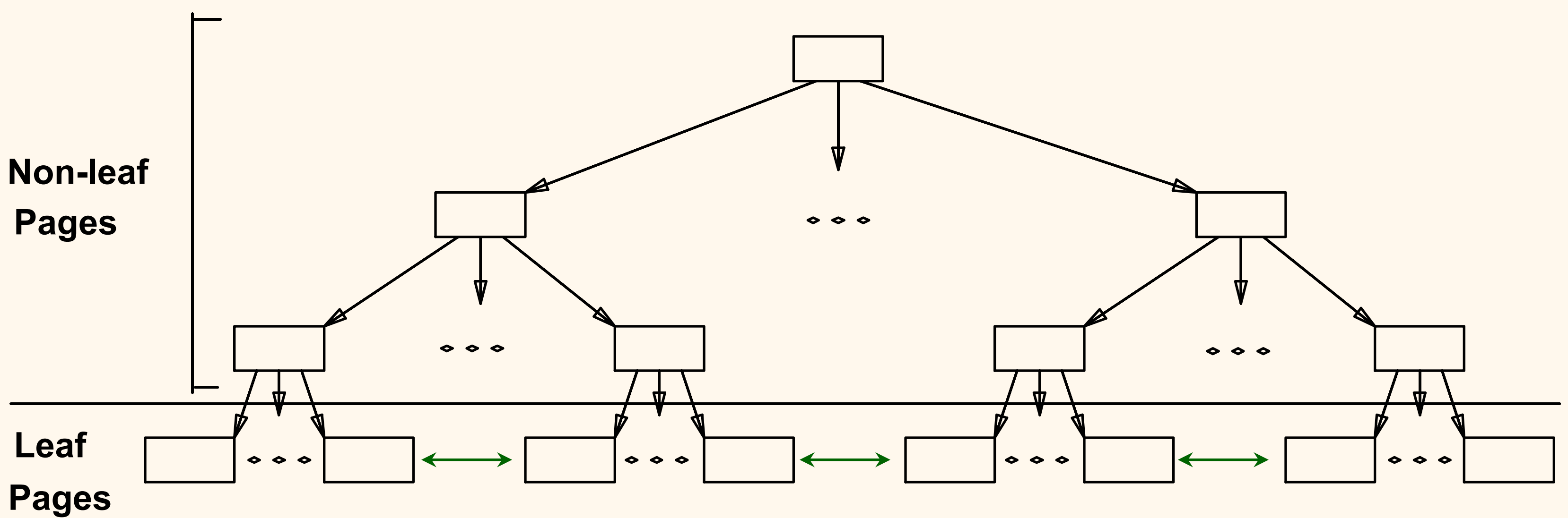
Hash-Based Indexes



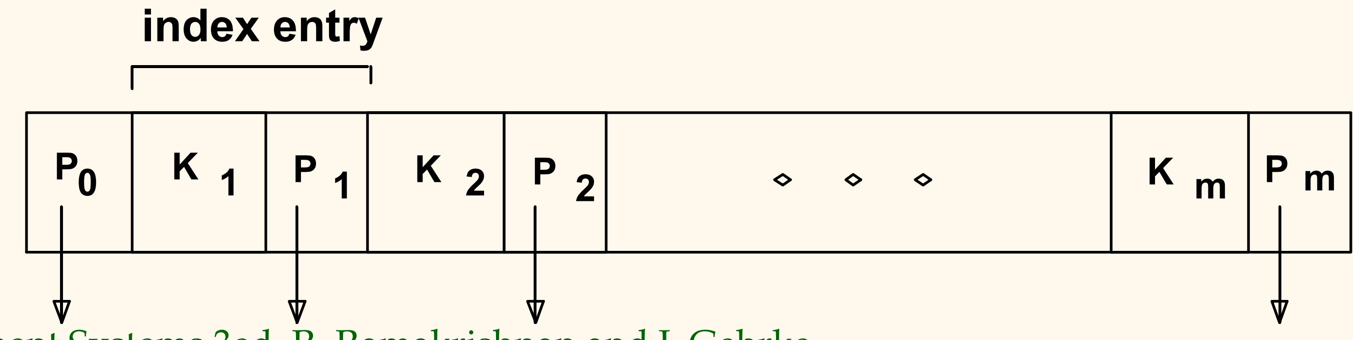
- Good for equality selections.
  - Index is a collection of *buckets.* Bucket = *primary* page plus zero or more *overflow* pages.
  - *Hashing function* **h**: **h**(*r*) = bucket in which record *r* belongs. **h** looks at the *search key* fields of *r.*
- If Alternative (1) is used, the buckets contain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

B+ Tree Indexes

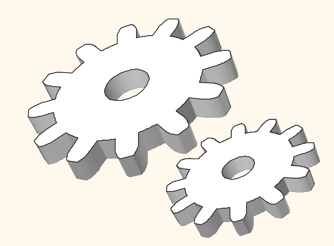


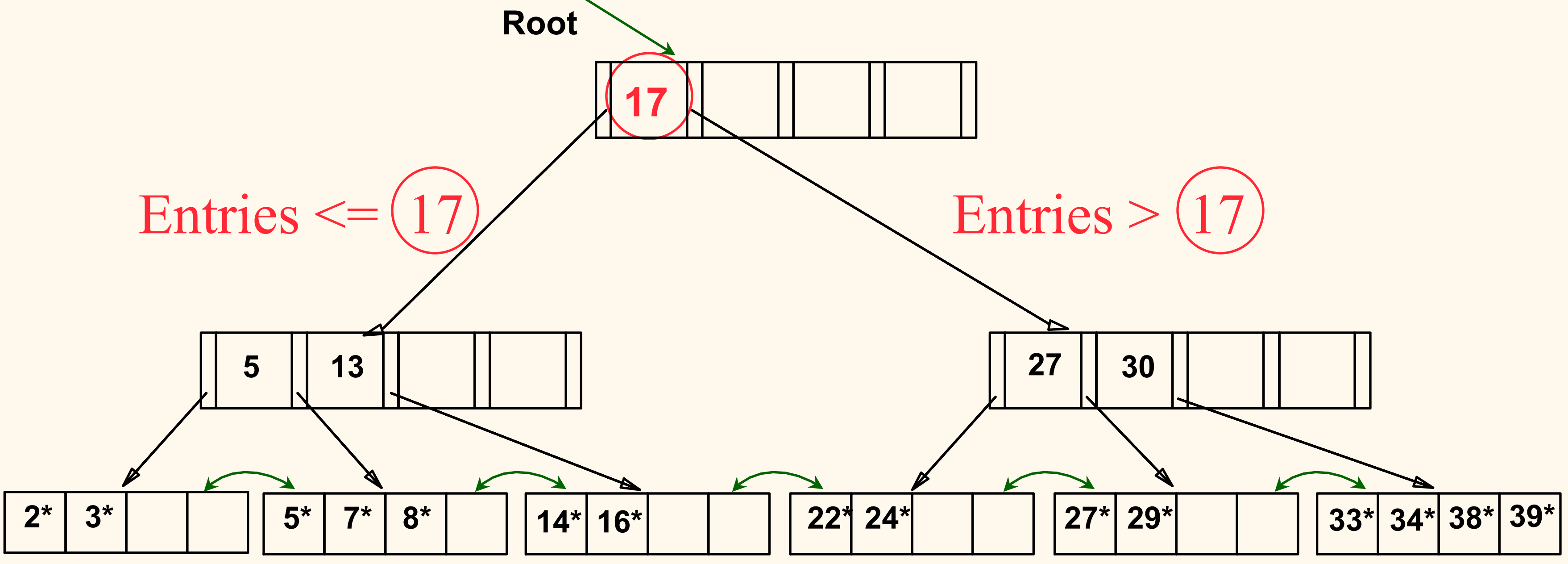


- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages contain *index entries*; they direct searches:



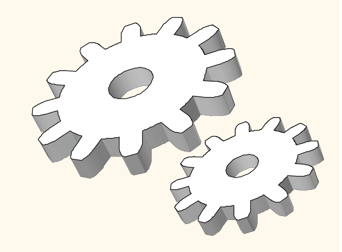
Example B+ Tree





- Find 28*? 29*? All > 17* 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

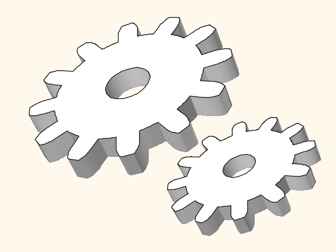


We ignore CPU costs, for simplicity:

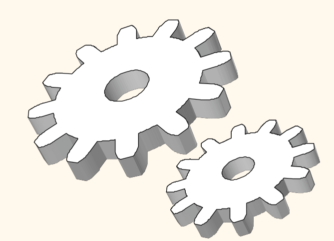
- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Average-case analysis; based on several simplistic assumptions.

☛ *Good enough to show the overall trends!*

Comparing File Organizations



- Heap files (random order; insert at eof)
- 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*>



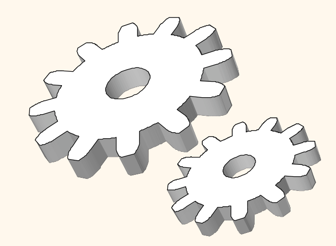
Operations to Compare

- Scan: Fetch all records from disk
- Equality search (e.g., "age = 30")
- Range selection (e.g., "age > 30")
- Insert a record
- Delete a record

Parameters of the Analysis

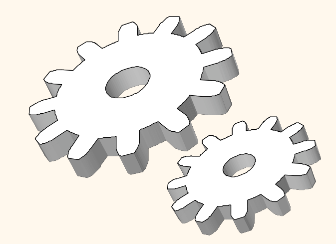
	B = # data pages	R = #records/ page	D = disk page I/O time	C = process single record	H = apply Hash function	F = index tree fan-out
Typical value			15 mlsec	100 nanosec	100 nanosec	100

Assumptions in Our Analysis



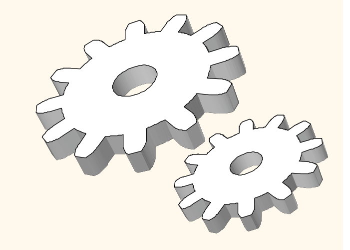
- Heap Files:
  - Equality selection on key; exactly one match.
- Sorted Files:
  - Files compacted after deletions.
  - Clustered files: pages typically 67% full.
  - ⇒ Total number pages needed = 1.5 B.
- Indexes:
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy.
    - ⇒ Index size = 1.25 B data size.
    - ⇒ #data entries/page = 10 (0.8R) = 8R.
  - Tree: 67% page occupancy of index pages (this is typical).
    - ⇒ #leaf pages = (1.5 B) 0.1 = 0.15 B.
    - ⇒ #data entries/page = 10 (0.67R) = 6.7R.

Scanning Cost



- Heap file: B(D + RC).
  - for each page (B)
  - Read the page (D)
  - For each record (R), process the record (C).
- Sorted File: B(D + RC).
  - Have to go through all pages.
- Clustered File: 1.5B (D+RC).
  - Pages only 67% full.
- Unclustered Tree Index: >BR(D+C). Bad!
  - for each record (BR)
  - retrieve page and find record (D + C).

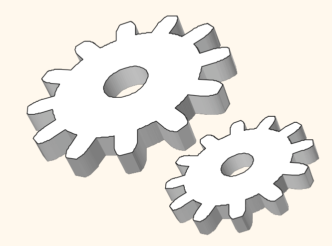
Exercise for Group Work



1. Estimate how long an equality search takes in (i) a heap file (ii) a sorted file (iii) a hash file, hashed on the search key, with at most one record matching the search key (i.e., the search is on a key field).

2. Estimate how long an insertion takes in (i) a heap file (ii) a sorted file (iii) a hash file.

	B = # data pages	R = #records/ page	D = disk page I/O time	C = process single record	H = apply Hash function	F = index tree fan-out
Typical value			15 msec	100 nanosec	100 nanosec	100

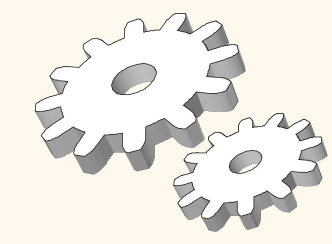


Cost of Operations

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

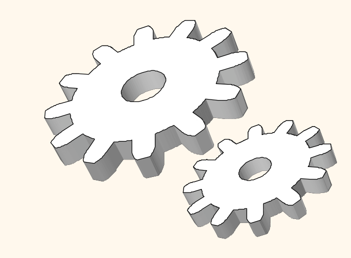
☛ *Several assumptions underlie these (rough) estimates!*

Index Illustrations



- Hash Insertion: 4 D I/Os: 2 to read/write data page, 2 to read/write index entry.
- Hash Index Illustration.
- Clustered Tree Index Illustration.

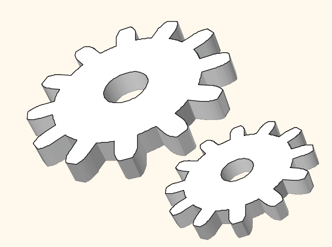
I/O Cost of Operations



	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	**BD**	**0.5BD**	**BD**	**2D**	**Search +D**
(2) Sorted	**BD**	**Dlog 2B**	**Dlog 2 B + # matches**	**Search + BD**	**Search +BD**
(3) Clustered Tree Index	**1.5BD**	**Dlog F 1.5B**	**Dlog 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(3 + 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!*
- *Order of magnitude results.*

Create Indexes in SQL-Server



- SQL Server supports many options for creating indices (more than we can cover).
- Sample Syntax:

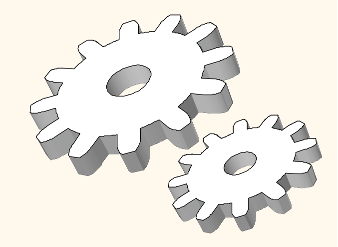
use aworks;

create index IX_Product_Color

on SalesLT.Product (Color);

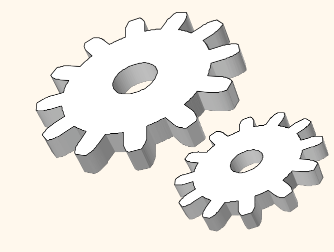
- More Examples

Understanding the Workload



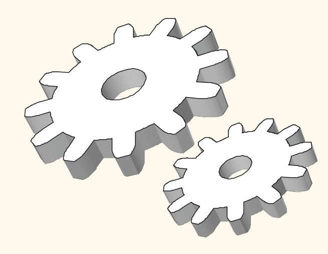
- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join 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



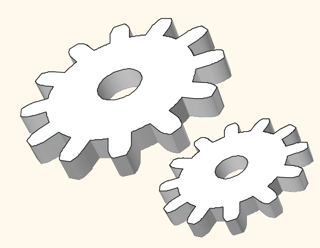
- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree?

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.

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. MS Index Tuning Wizard

Examples of Clustered Indexes

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

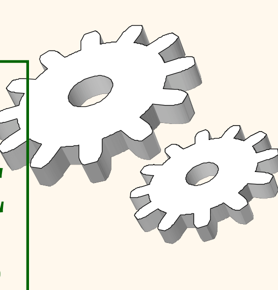
- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?

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

- 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.hobby=Stamps
```

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>

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

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

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

<E.dno>

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

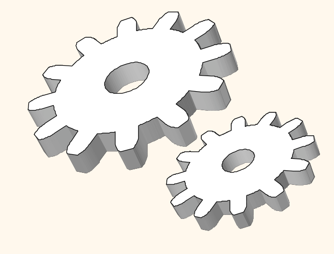
<E.dno,E.sal> *Tree index!*

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

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

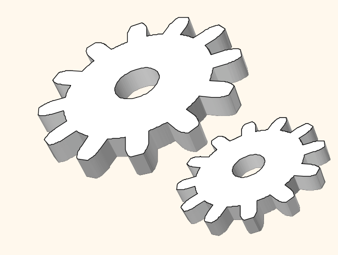
```
SELECT AVG(E.sal)
FROM  Emp E
WHERE  E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```

Summary



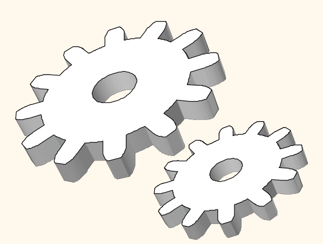
- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values.

Summary (Contd.)



- Data entries can be actual data records, <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 (Contd.)



- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates? What attributes/relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision, demanding on DBMS but potentially high payoff.