# RAID in Practice,
# Overview of Indexing

CS634
Lecture 4, Feb 04 2014

Slides based on “Database Management Systems” 3rd ed, Ramakrishnan and Gehrke

# Oracle on RAID

As most Oracle DBAs know, rules of thumb can be misleading but here goes:

- If you can afford it, use RAID 1+0 for all your Oracle files and you shouldn't run into too many problems. If you are using ASM, use this RAID level for all LUNs presented to ASM. (ASM = automatic storage management)
- To reduce costs move datafiles with lower access rates to RAID 5. If you are using ASM, this may mean defining multiple disk groups to hold different files.
- To reduce costs further move the other datafiles to RAID 5.
- To reduce costs further experiment with moving redo logs and rollback/undo tablespaces to RAID 5. Pick the combinations that work best for your system.
- If cost dictates it, move all your Oracle files on to RAID 5.

Ref: https://oracle-base.com/articles/misc/oracle-and-raid

# Disks and Files: RAID in practice

For a big enterprise database: RAID 5 or RAID 10 (or 50 or 6 or 60)

Example 32 disks in one box, with room to grow (disk array pic)

- This RAID enclosure can hold up to 96 disks.
- Starter system with 8 disks, controller ~$18,000
- Uses 15Krpm disks, twice 7200rpm.
- Disks are 146GB, 300GB, …, 1TB each.
- Features automatic failover, rebuild on spare

- Why ever use these little 146GB disks?

# High-end RAID Example, continued

Why use small disks in enterprise RAID?

- Each disk, of any size, provides about 100ops/sec at 7200rpm, 200 ops/sec at 15Krpm.

- Many apps need quick access to small data sets, so the important performance measure is total ops/sec.
- So small disks are fine, and cheaper, and faster to rebuild the replacement when crashed.

- 30 disks here means 30*200 = 6000 ops/sec. Here keeping 2 as spares…

# High-end RAID Example, continued

Using small disks in enterprise RAID

- RAID 5: suppose 6 disks/RAID, 5 RAIDs, so useful storage = 5/6 * 30*146GB = 3.6TB
- RAID 10: storage = ½ * 30 * 146GB = 2.2 TB

- Either way, read rate = 6000 ops/sec (see last slide), write rate = 3000 ops/sec

- Rebuild time:
- RAID 5: need to read 5 disks, write 1
- RAID 10: need to read 1 disk, write 1

# Gluing RAIDs together

- As we will see, databases know how to use multiple disks/files as one big disk resource
- We can also do the gluing together at the hardware or OS level using the JBOD (just a bunch of disks) capabilities. Instead of disks, you can use whole RAIDs.

# Low-end RAID Example

For a research project, or web startup, want something cheaper…

Software RAID: OS drives ordinary disks

• Linux and Windows can do RAID 0, 1 in software.
• Linux can do software RAID 5, Windows Server has a similar option.
• Linux has its own “RAID 10” scheme, different from traditional RAID 10, but claims similar characteristics. Windows can do regular RAID 10 Ref

# Example of Software RAID

- 16 7200rpm disks of 200GB each for say $80 each, total $1300
- 16-port disk controller ~$400

- Build 2 RAID arrays 6 disks each, keeping 4 spares.
- Database can span the multiple RAIDs easily.

- End up with 12*100 ops/sec capability at <$2000
- Why not one big RAID? Takes too long to rebuild.
- Be ready to add another RAID to expand.

# Hardware RAID

Instead of a “plain” disk controller, get a RAID controller, AKA disk array controller.

End up with “hardware RAID”, looks like one big disk to OS.

A 16-port RAID controller can cost $1500, provides higher performance and system crash handling:

- Provides a cache to speed up reads and writes,
- Has battery backup or capacitors to power the cache while it saves its state to SSD or disk.
- The SSD here is small, just big enough to hold the data in flight

# Hybrid SSD/HDD Solutions

- SSD: no seek time/rotational delay, so much faster
- HDD: much cheaper

So put these together in one hybrid disk: SSHD

- Now available for desktop systems as well as enterprise servers.
- Seagate ref on hybrid drives.

Hybrid arrays: Wikipedia list

- On that list: Oracle Exadata X6-2 :"X6-2 High Capacity storage servers contain 12 disks, 8TB each, for a total of 96 terabytes of raw storage capacity. To improve I/O response times, High-Capacity storage servers also employ 12.8 terabytes of PCIe flash to cache active data blocks."
  - Here size of SSD is about 15% of the size of the HDD (typical)
  - Scan rate 512 GB/s, read iops 4.7 M ops/s, write iops 4.1 M ops/s (8KB/op, i.e. one DB block/op)

# Hybrid SSD/HDD: how does it work?

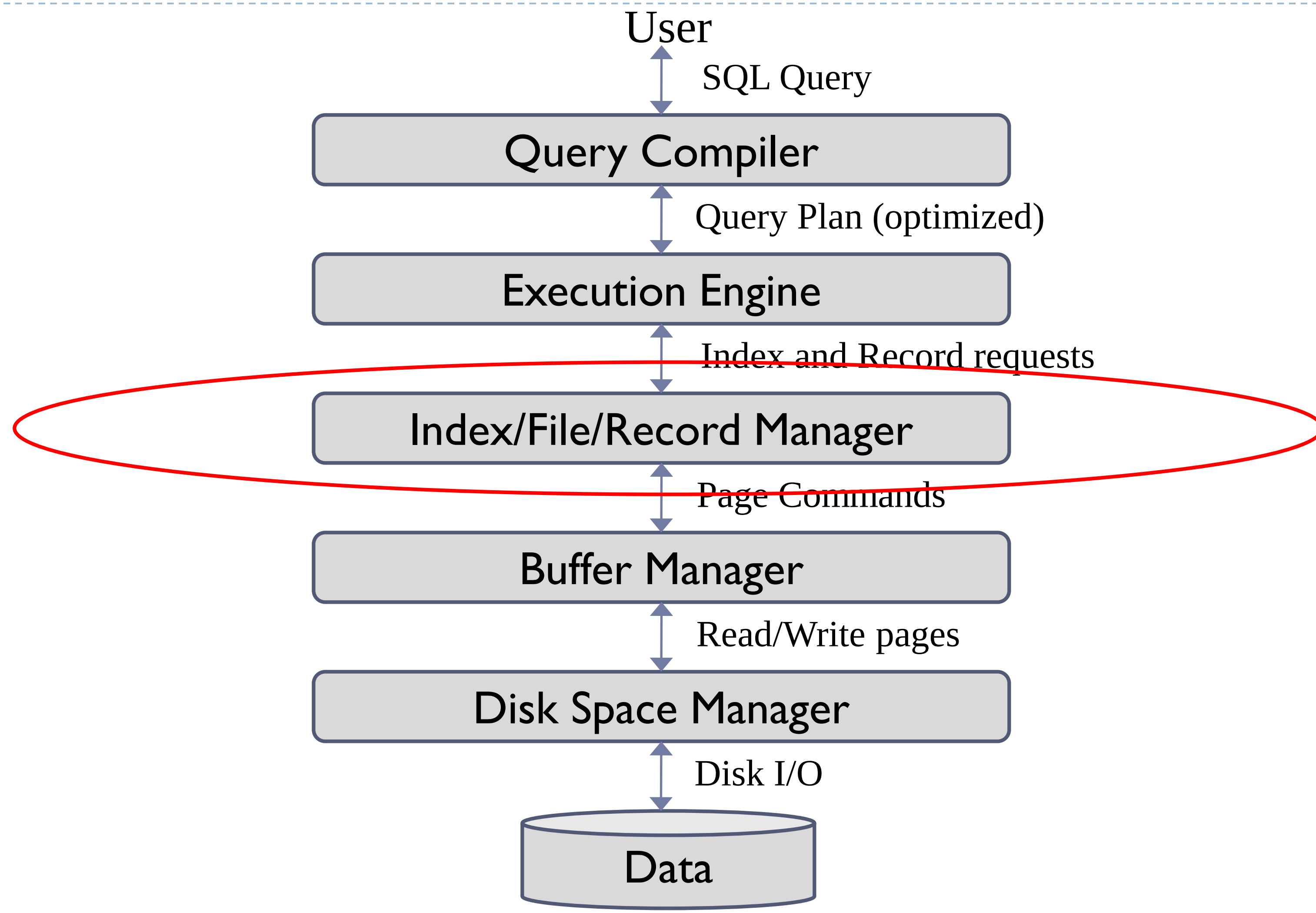
The drive or array tracks usage of blocks of data

- High usage: move to SDD (possibly with backup to HDD)
- Low usage: move to HDD only

Caching is well understood, so this is just a new application of old principles

  - Based on 80-20 rule or 90-10 rule: 80% of refs are to 20% of data or even 90% of refs are to 10% of data.
  - Again, key algorithm is replacement policy.

# On to Chapter 8: Intro to Indexing




A first course in database systems, 3rd ed, Ullman and Widom

# Data Organization

- Fields (or attributes) are grouped into records
  - In relational model, all records have same number of fields
  - Fields can have variable length
  - Records can be fixed-length (if all fields are fixed-length) or variable-length
- Records are grouped into pages
- Collection of pages form a file
  - Do NOT confuse with OS file
  - This is a DBMS abstraction, but may be stored in an OS file or multiple files or a "raw partition"

# Files of Records

- Page or block access is low-level
  - Higher levels of DBMS must be isolated from low-level details

- FILE abstraction
  - collection of pages, each containing a collection of records
  - May have a "header page" of general info
  - May contain table data or index data or …, whatever the DB needs

- File operations
  - read/delete/modify a record (specified using *record id*)
  - insert record
  - scan all records

# Files of Records

May be organized in several ways:

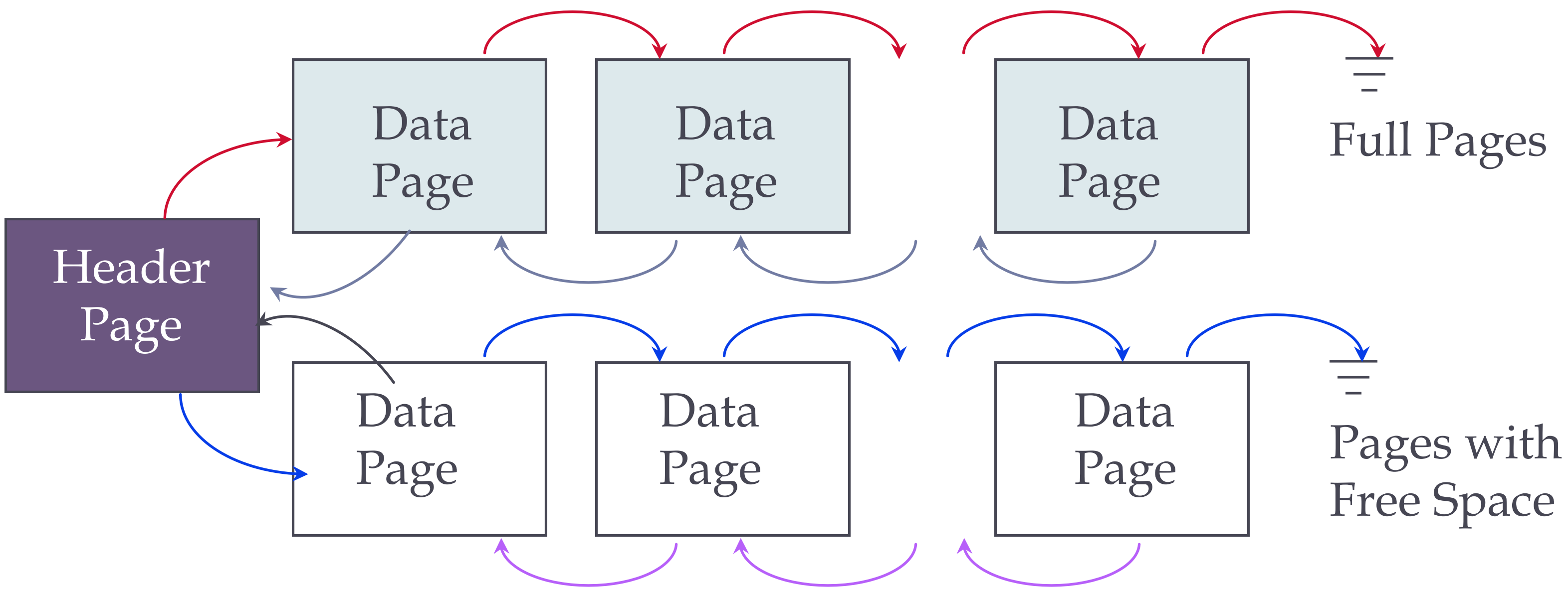
- Heap files: no order in data records
  - Intro p. 276, Covered in Sec. 9.5.1, and following slides
- Sorted file: data records have a key, and records are in that key order (hard to maintain, so rarely used)
  - Covered in Sec. 8.4.
- Clustered file: data records have a key, and records are pretty much in that key order (more practical)
  - Intro p. 277, more in Sec. 8.4.4
- Index file: records are "data entries", several types exist
  - Intro, pg. 276

# Unordered Files: Heap

- Heap
  - simplest file structure
  - contains records in no particular order
  - as file grows and shrinks, disk pages are allocated and de-allocated

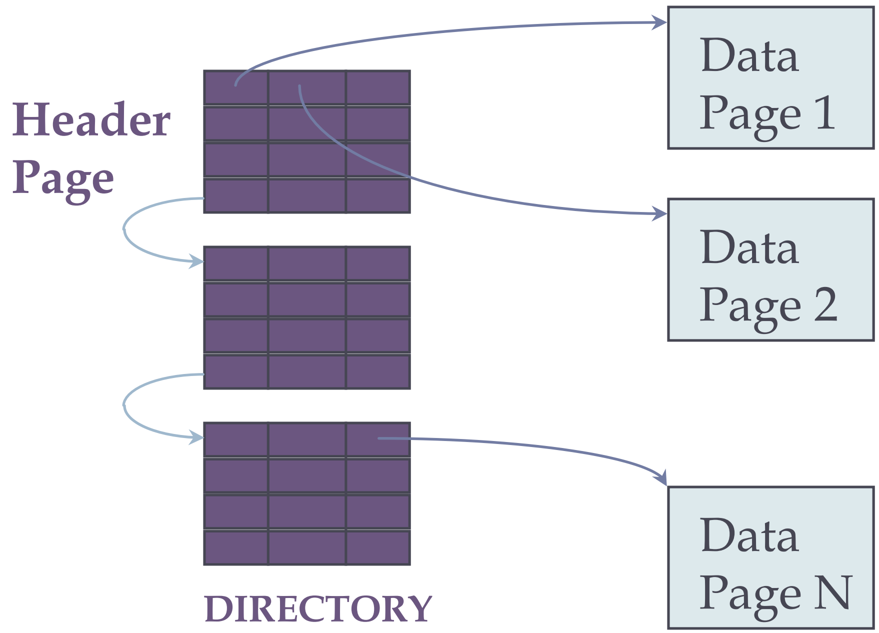
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page

# Heap File Implemented as a List



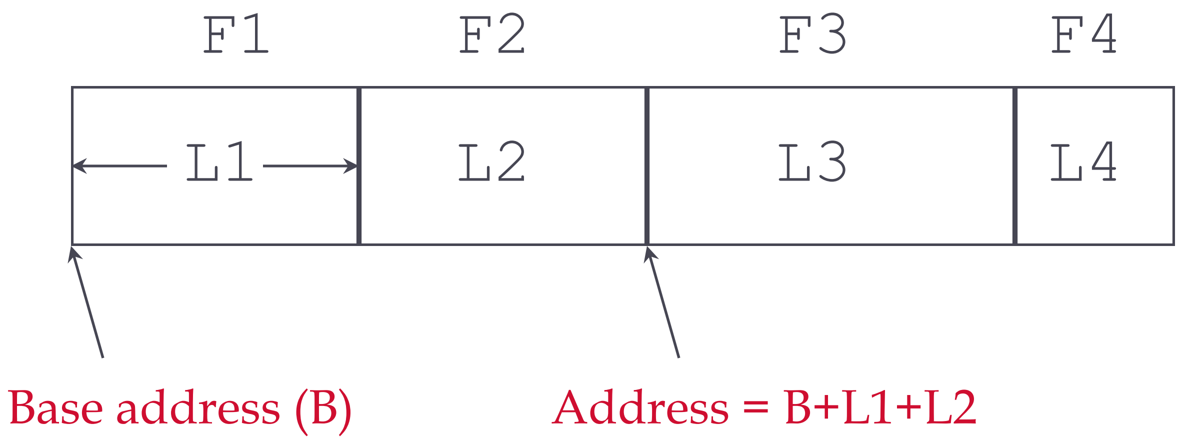

Not a great idea, not used by Oracle, mysql, other serious DBs

# Heap File Using a Page Directory




- Page entry in directory may include amount of free space
- Directory itself is a collection of pages
  - linked list implementation is just one alternative

# Record Formats: Fixed Length

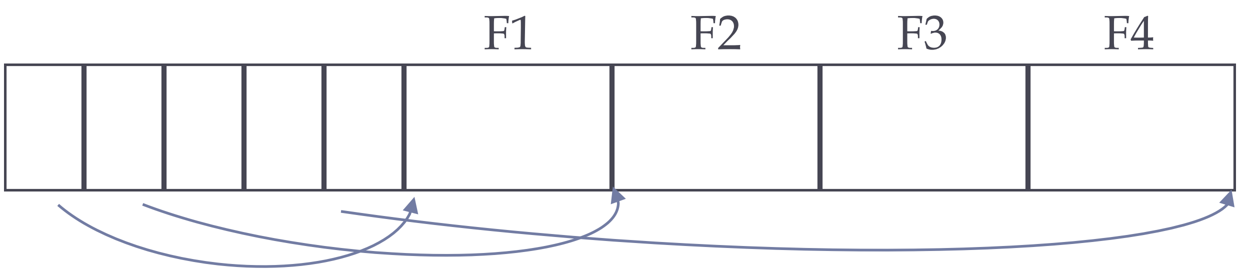



- Information about field types same for all records in a file; stored in *system catalogs*.
- Finding *i'th* field does not require scan of record.

# Record Formats: Variable Length

- Two alternative formats (# fields is fixed):

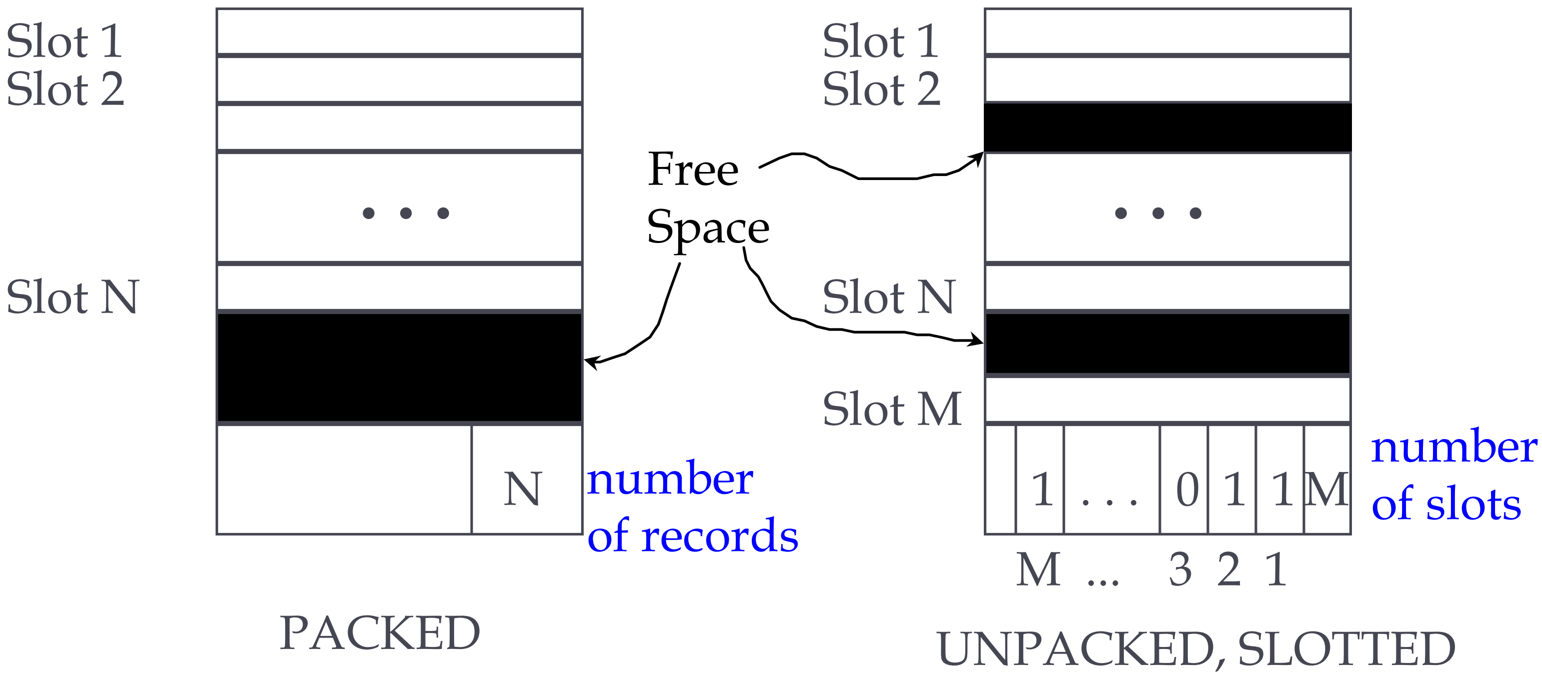

Fields Delimited by Special Symbols




Array of Field Offsets

Second offers direct access to i'th field, efficient storage of *nulls*; small directory overhead. Ignore first format.

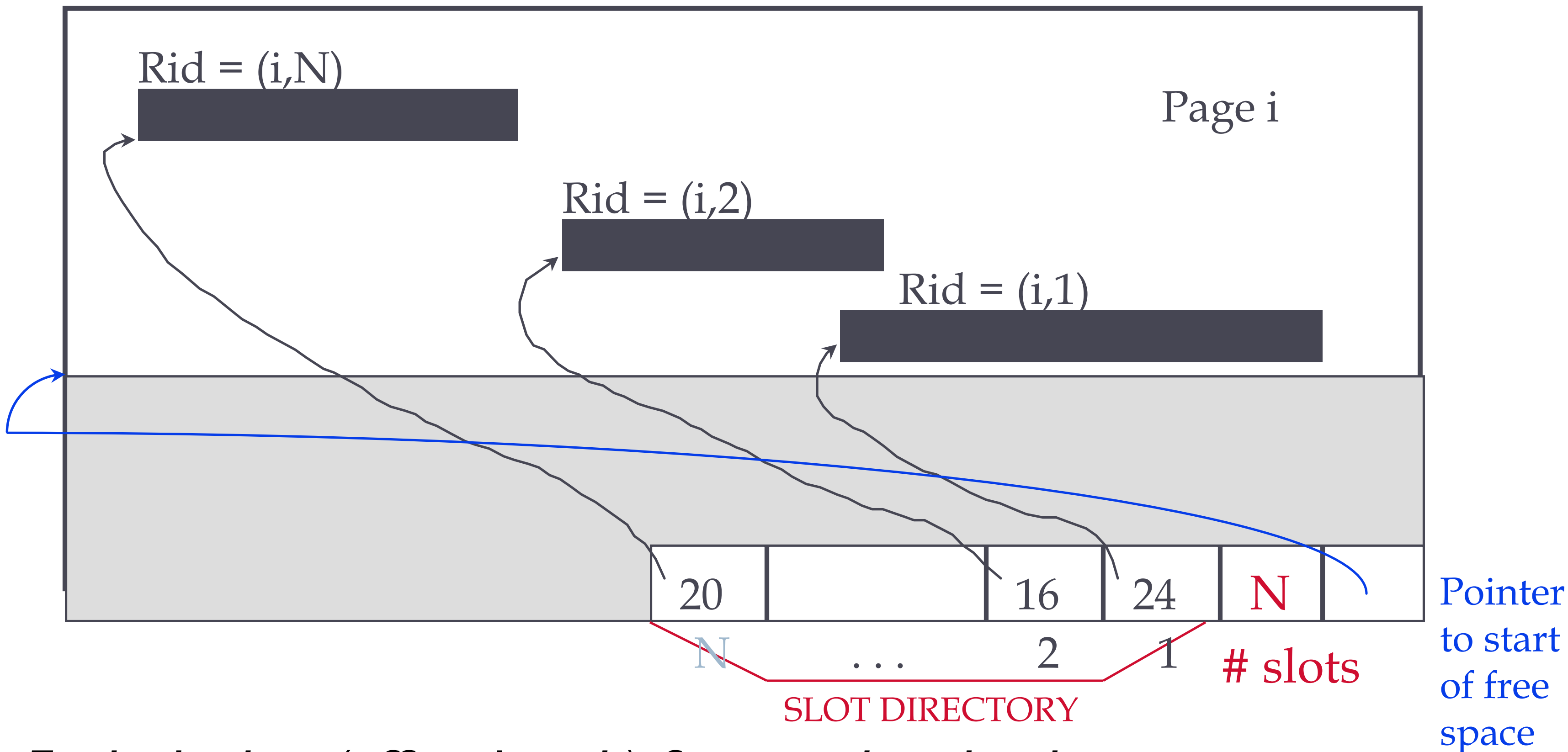
# Page Formats: Fixed Length Records




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

See next slide for the usual row format for both fixed and variable-length records.

# Page Formats: Variable Length Records




*Each slot has (offset, length) for record in slot directory.*

*Can move records on page without changing rid; so, attractive for fixed-length records too.*

# Summary

- Disks provide cheap, non-volatile storage
  - Random access, but cost depends on location of page on disk
  - Important to arrange data sequentially to minimize *seek* and *rotation* delays
- Buffer manager brings pages into RAM
  - Page stays in RAM until released by requestor
  - Written to disk when frame chosen for replacement
  - Choice of frame to replace based on *replacement policy*
  - Tries to *pre-fetch* several pages at a time
- Data stored in file which abstracts collection of records
  - Files are split into pages, which can have several formats

# Data Organization (review)

- Index/File/Record Manager provides abstraction of file of records (or short, file)
  - File of records is collection of pages
  - I/F/R Manager also referred to File and Access Method layer, or short, File Layer
- File operations
  - read/delete/modify a record (specified using *record id*, *AKA rid*, *Oracle ROWID*)
  - insert record
  - scan all records
- Record id functions as data locator
  - contains information on the address of the record on disk
  - e.g., page in file and directory slot number in page
  - Ready for random access on disk, no real search

# File Organization

1. Unsorted, or heap file
   - Records stored in random order

2. Sorted according to set of attributes
   - E.g., file sorted on <*age*>
   - Or on the combination of <*age*, *salary*>

- No single organization is best for all operations
  - E.g., sorted file is good for range queries
  - Example: select * from T where key > 100 and key < 200
  - But it is expensive to insert records
  - We need to understand trade-offs of various organizations

# Oracle Files and Tablespaces

- Oracle uses a "file" concept, which can refer to a file or a raw partition, i.e. a low-level OS page container.
- An Oracle tablespace consists of one or more files combined to make a file-like page container.
- Tablespaces contain tables and indexes.
- Thus when the book says File, think Oracle tablespace.
- To expand a tablespace, can add a new file to it.
- We can build tablespaces across multiple disks.

# Oracle ROWIDs

- The Oracle ROWID format, the "extended ROWID" form, is displayed as a string of four components
- Layout, with letters in each component representing a base-64 digit: (file# is relative to tablespace)

  object# file# block#-in-file slot#-in-block

  ```
  OOOOOOFFFBBBBBBRRR
  AABio6AAHAAAWwcAAB
  AABio6|AAH|AAAWwc|AAB
  ```

- Base 64: A..Za..z0..9+/ (A = 0, B=1, … + = 62, / = 63)
- 64 = 2^6, so 6 bits each, 18 chars, means 108 bits total, or 13.5 bytes. Some internal RIDs may be shorter than this.

# Oracle ROWIDs

You can use pseudo-column ROWID to display these

```
SQL> select sname, rowid from sailors;
SNAME                    ROWID
------------------------ -----------
jones                    AACHzYAAHAAANxnAAA
jonah                    AACHzYAAHAAANxnAAB
ahab                     AACHzYAAHAAANxnAAC
moby                     AACHzYAAHAAANxnAAD
```

- We see these rows are all on the same block, or page, of file AAH = 7, block AAANxn = 13*64^2+(26+23)*64+(26+13)
- Mysql does not expose its RIDs. This is an Oracle-specific feature, not part of SQL-92 or later standards.

# Index Basics

Example Table: sailors(sid, sname, rating, age)

Create an index on sname and use it in a query:

```
SQL> create index snamex on sailors(sname);
Index created.

SQL> select * from sailors where sname='ahab';
       SID SNAME                    RATING        AGE
---------- ---------------- ---------- ----------
        22 ahab                      7         44
```

Here the index speeds up queries that need to find certain values of sname in the table.

The index is named snamex, and its search key is sname.

It is associated with table sailors.

# Index Basics

Example Table: sailors(sid, sname, rating, age)

Create an index on sname:

```
SQL> create index snamex on sailors(sname);
```

- The index is named snamex, and its search key is sname.
- Its lowest-level contents look like this: (Oracle)

| sname | ROWID |
|---|---|
| ahab | AACHzYAAHAAANxnAAC |
| jonah | AACHzYAAHAAANxnAAB |
| jones | AACHzYAAHAAANxnAAA |
| moby | AACHzYAAHAAANxnAAD |

- Note how the sname values are now in sorted order. There is some additional structure used to guide access to these "data entries".

# Indexes

- Structures that speed up operations
  - Improve performance with some (small) storage overhead
- Sorted file can have only one sort order, e.g., age
  - But what if we also need to support range queries on salary?
  - We can build index on salary!
- Two varieties of index structures
  - Tree-based: best for range queries, also support exact match
  - Hash-based: best for exact-match queries
    - No support for other queries
- Also bitmap indexes, not covered in Chap 8-12

# Index Properties

- Provides "search-by-content" of a certain table
  - Given search key, return rid or rids in the table
  - For example, given 'ahab', return RID for that row in sailors

- An index has *search key fields,* subset of fields of its table
- For example, the index snamex has search key field sname, one of the columns of table sailors.
  - Any field subset in the table can be the search key
  - *Do not confuse term with primary key!*

# Index Properties

- Index contains collection of *data entries*
- A data entry for key value k contains enough info to locate one or more table rows matching k in the search key columns.
- For ex, the data entry for ‘ahab’ could be (‘ahab’, RID)

- A data entry for k is denoted *k** in the text.
- so here k=‘ahab’, k* = (‘ahab’, RID)

- But not all data entries look like this. In some indexes, the whole row (AKA data record) is held in the data entry.

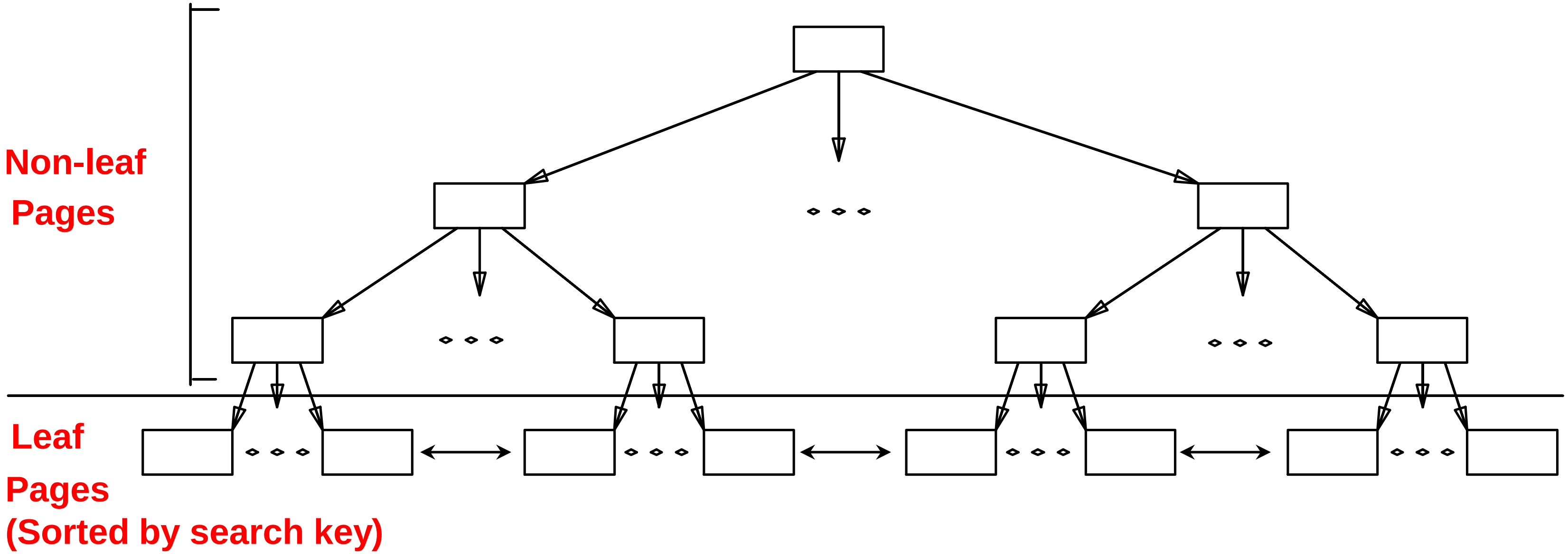
# Index Properties

- Example Table: sailors(sid, sname, rating, age)
- Example Index: on sname

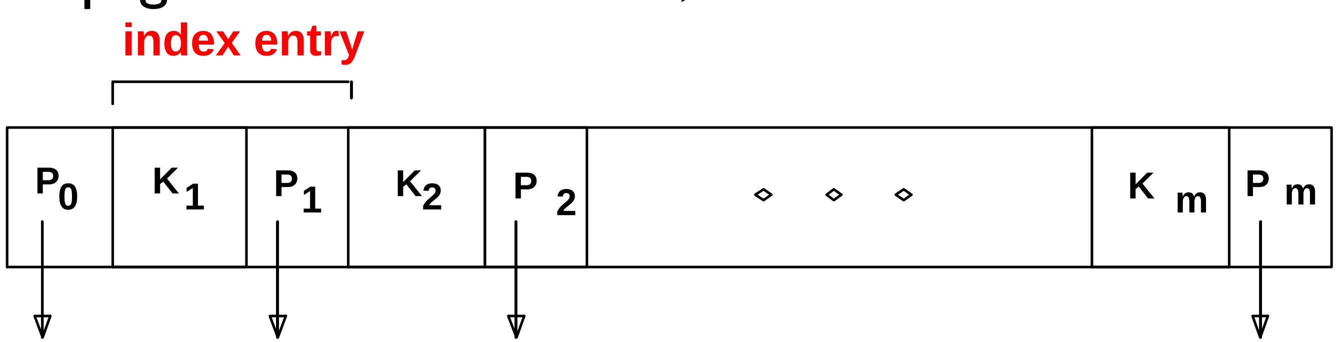
- One way, the data entry for 'ahab' could be ('ahab', RID)
- so here k='ahab', k* = ('ahab', RID)

- But not all data entries look like this. In some indexes, the whole row (AKA data record) is held in the data entry.
- Then k='ahab', k* = (22,'ahab',7,44.0) with known key.
- This is alternative 1 on pg. 276, and above ex. is Alt. 2.

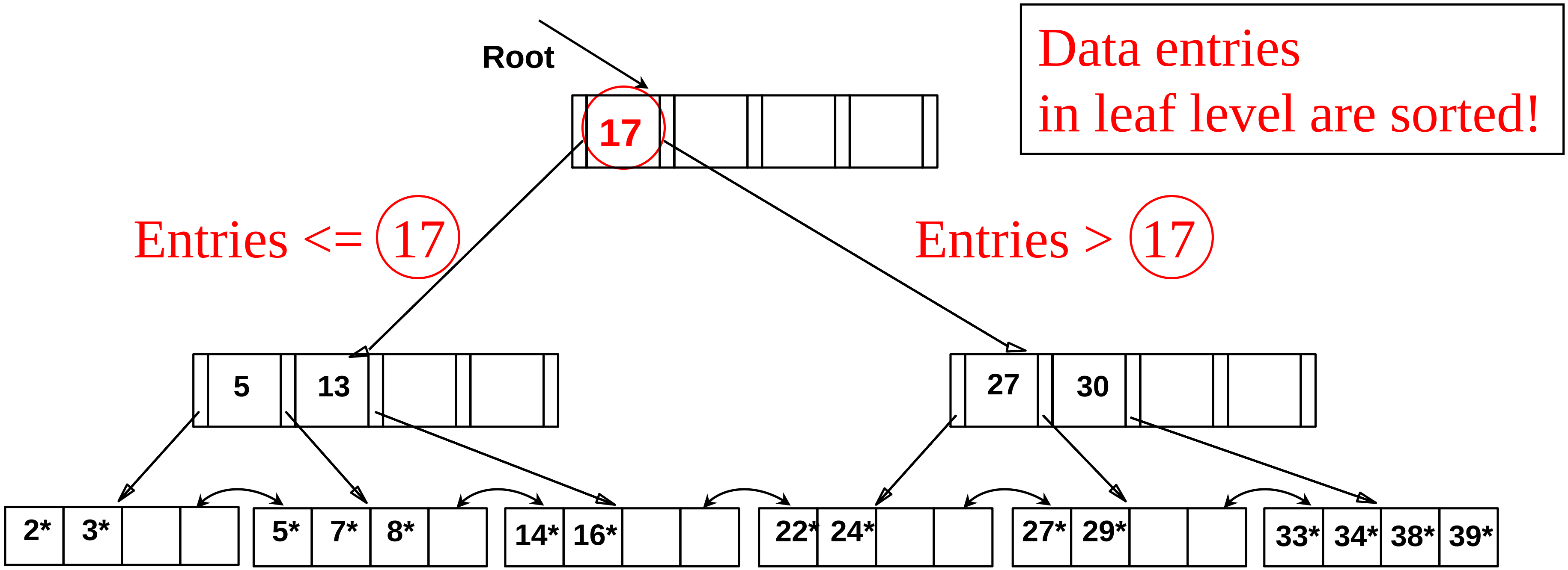
# Tree Index Example




- Leaf pages contain *data entries*, and are chained
- Non-leaf pages have *index entries*, used to direct search

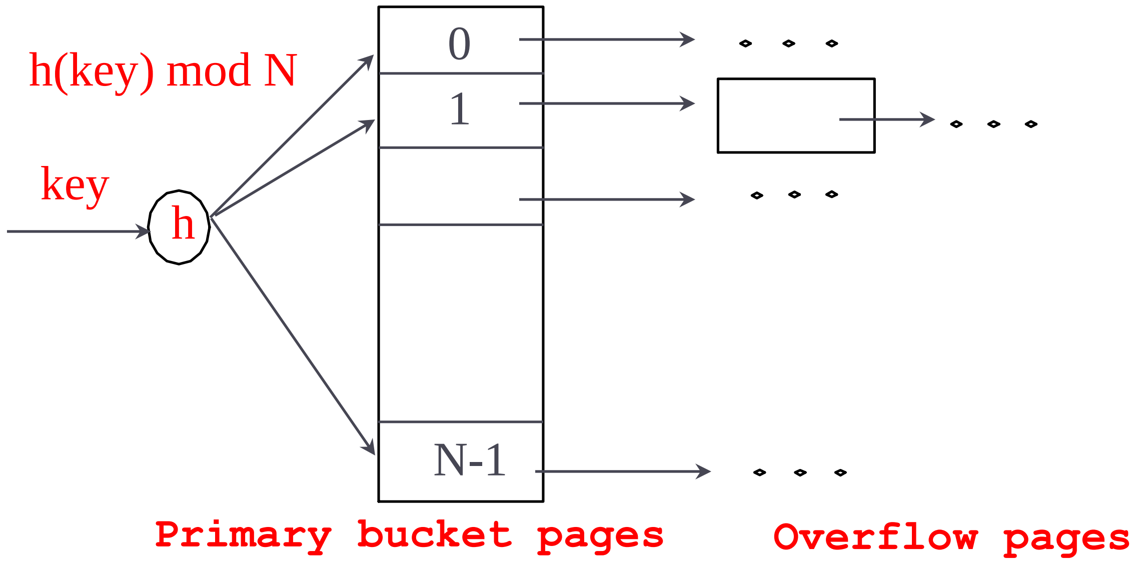


# Search with B+ Tree




Supports efficiently Exact-Match and Range queries on search key

# Hash Index Example




- Buckets represent *index entries*, *data entries* look the same as in the case of tree index
- The strength of the method relies in the capacity of function *h* to distribute data uniformly

# Alternatives for Data Entry k* in Index

1. **Data record** with key value *k*
   - Leaf node stores actual record
   - Example: the sname index we looked at earlier: k* = (22,'ahab',7,44.0)
   - Only one such index can be used (without duplication of table data)
2. *<k, rid>* rid of data record with search key value *k*
   - Only a pointer (rid) to the page and record are stored
   - Example: the sname index we looked at earlier: k* = ('ahab', RID)
3. *<k, list of rids>* list of rids of records with search key *k*
   - Similar to previous method, but more compact
   - Disadvantage is that data entry is of variable length
   - Can be considered a compressed version of 2.

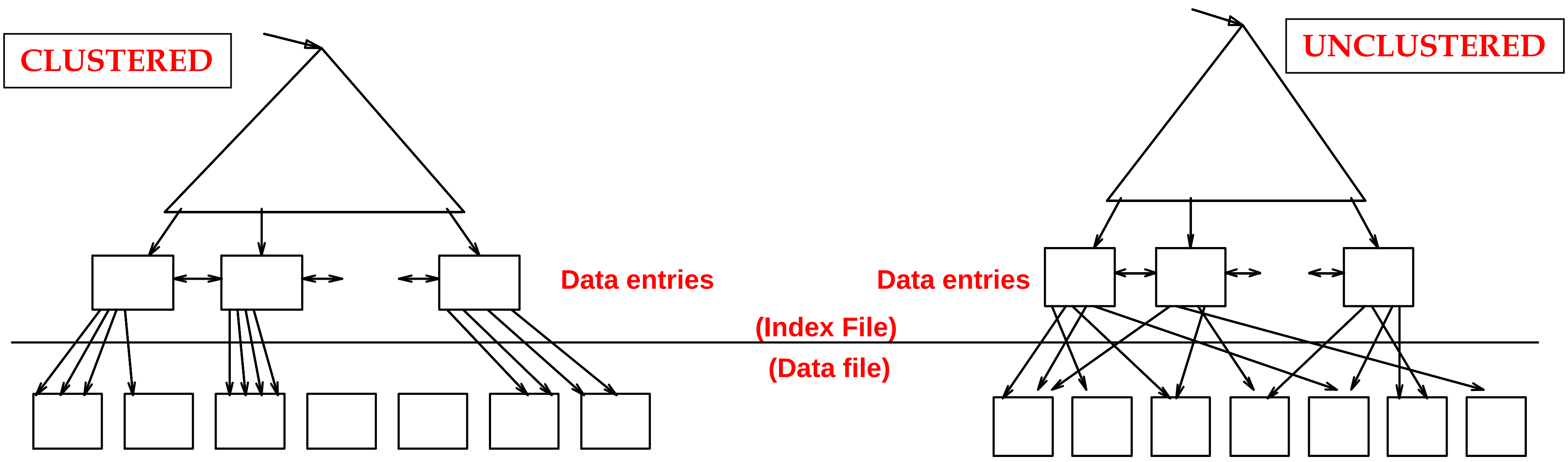
- Several indexes with alternatives 2 and 3 may exist

# Index Classification

- *Primary vs. secondary*
  - if search key contains primary key, then it is called primary index
  - *Unique* index: Search key contains a candidate key

- *Clustered vs. unclustered*
  - If order of data records is close to order of data entries, then the index is clustered; Alternative 1 is clustered by definition
  - In practice, sorted files are rare, so alternative 1 is the choice; also called a clustered file organization
  - A file can be clustered on at most one search key
  - Clustered indexes behave much better for ranges and scans

# Clustered vs. Unclustered Index




- To build clustered index, first sort the heap file, leaving some free space on each page for future inserts
- Overflow pages may be needed for inserts
  - Hence order of data records is close to the sort order

# Clustered Indexes in Practice

- Oracle doesn't have general clustered indexes
- It has "index organized tables" and "table clusters" that have some similar characteristics
- If the table will have few updates, you can sort the load data, load the table and it will be effectively clustered.
- Partitioning has a similar effect of grouping same-key data together, well supported in Oracle.

- Mysql also does not have general clustered indexes
- It makes a clustered index on the primary key.
- That's usually fine, but sometimes we would like the table clustered by a non-unique key, say zipcode.
- Mysql also supports partitioning.

- DB2 and SQL Server have clustered indexes and partitioning.