CS660: Intro to Database Systems

Class 5: File Organization & Indexing

Instructor: Manos Athanassoulis

https://bu-disc.github.io/CS660/

File Organization & Indexing

Page Layout (NSM, DSM, PAX)

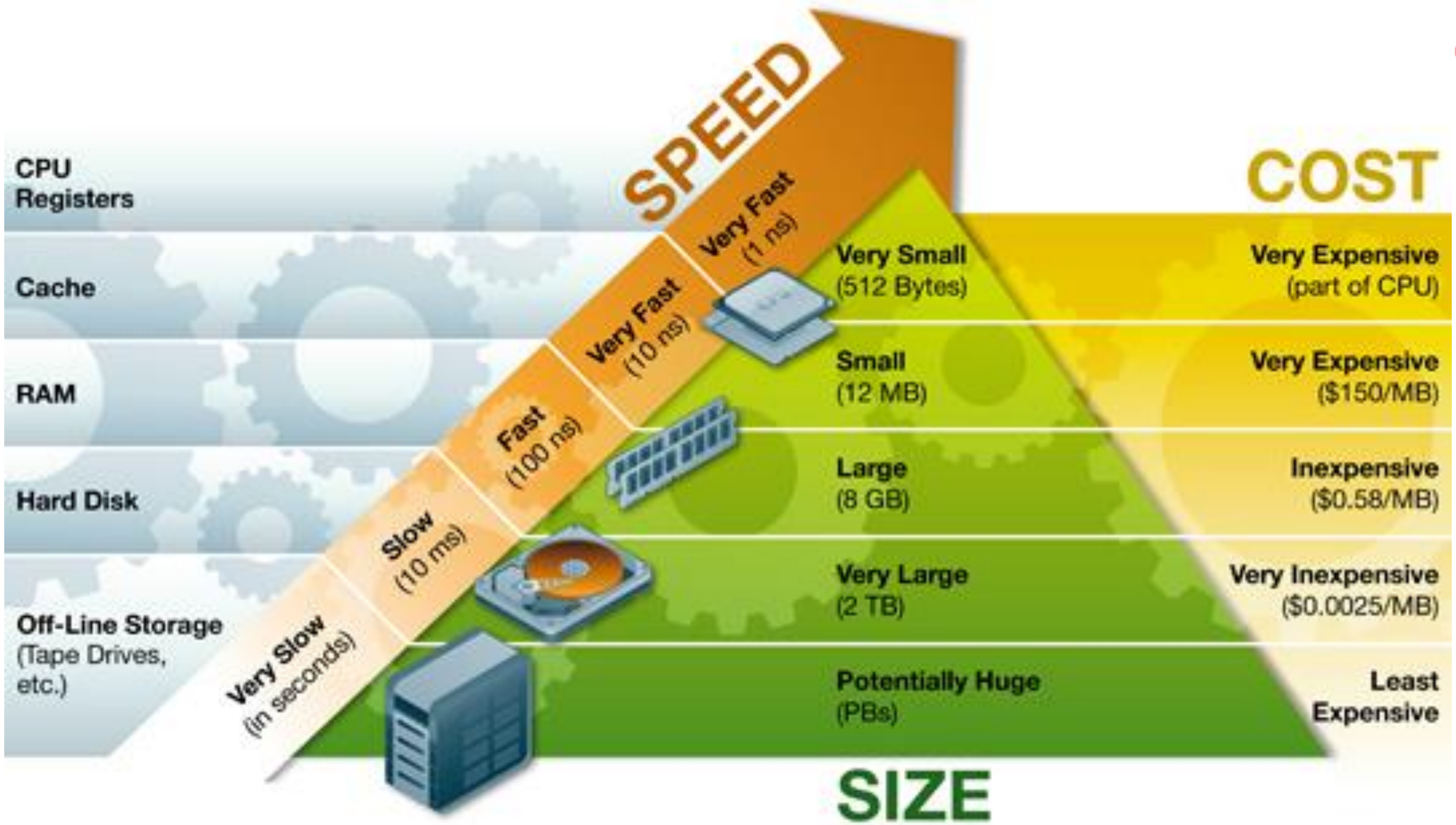
Readings: Chapter 8.1, 8.4, 9.6, 9.7, PAX paper

File organization (Heap & sorted files)

Index files & indexes

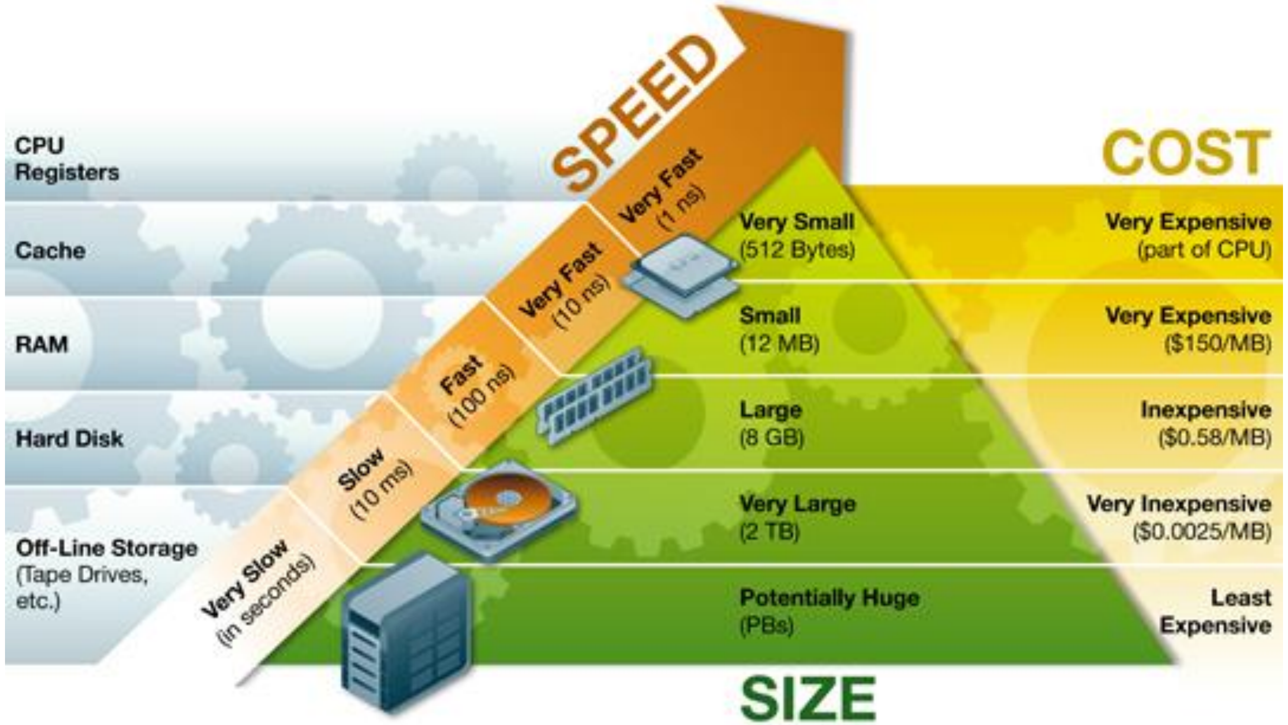
Index classification

Storage Hierarchy



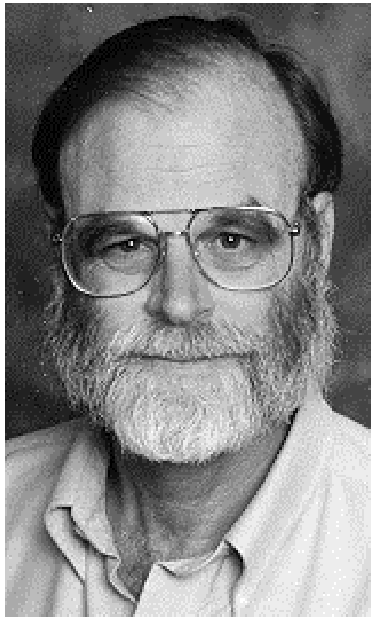
Memory, Disks

- Storage Hierarchy:
- cache, RAM, disk, tape, …
  - RAM is (usually) not enough
- Unit of buffering in RAM:
  - "Page" or "Frame"
- Unit of interaction with disk:
  - "Page" or "Block"



- "Locality" and sequential accesses → good disk performance
- Buffer pool management
  - <u>Slots</u> in RAM to hold Pages
  - <u>Policy</u> to move Pages between RAM & disk

memory hierarchy (by Jim Gray)



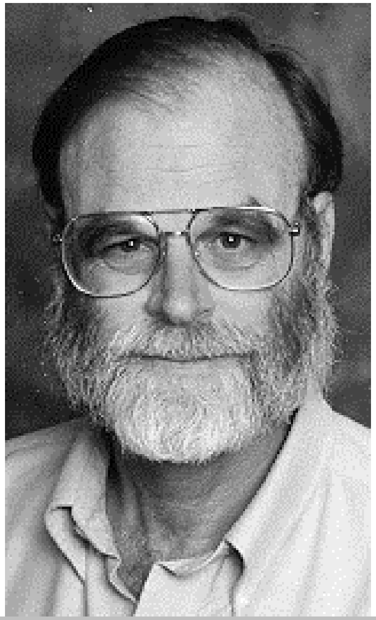
Jim Gray, IBM, Tandem, Microsoft, DEC

ACM Turing Award 1998

ACM SIGMOD Edgar F. Codd Innovations award 1993

	registers/CPU	my head ~0
2x	**on chip cache**	this room 1min
10x	**on board cache**	this building 10min
100x	**memory**	Washington, DC 5 hours
10^6x	**disk**	Pluto 2 years
10^9x	**tape**	Andromeda 2000 years

memory hierarchy (by Jim Gray)



	registers/CPU	my head ~0
2x	**on chip cache**	this room 1min
10x	**on board cache**	this building 10min

45TB @ $150

tape?
sequential-only magnetic storage
still a multi-billion industry

Today: File Storage

Transfer unit of a page is OK when doing I/O, but higher levels of DBMS operate on *records* and *files of records*

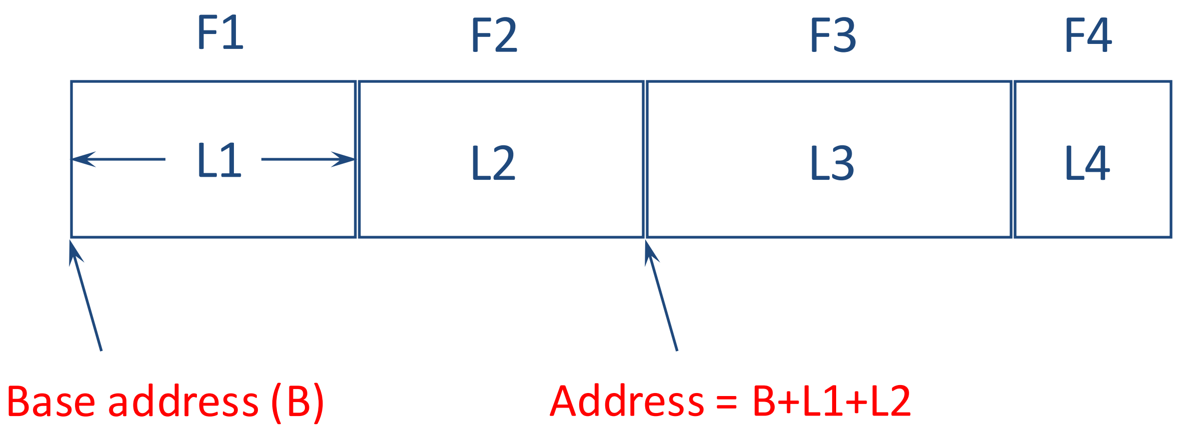
Next topics

organize **records** within pages

keep **pages** of records on disk

support **operations on files** of records efficiently

Record Formats: Fixed Length

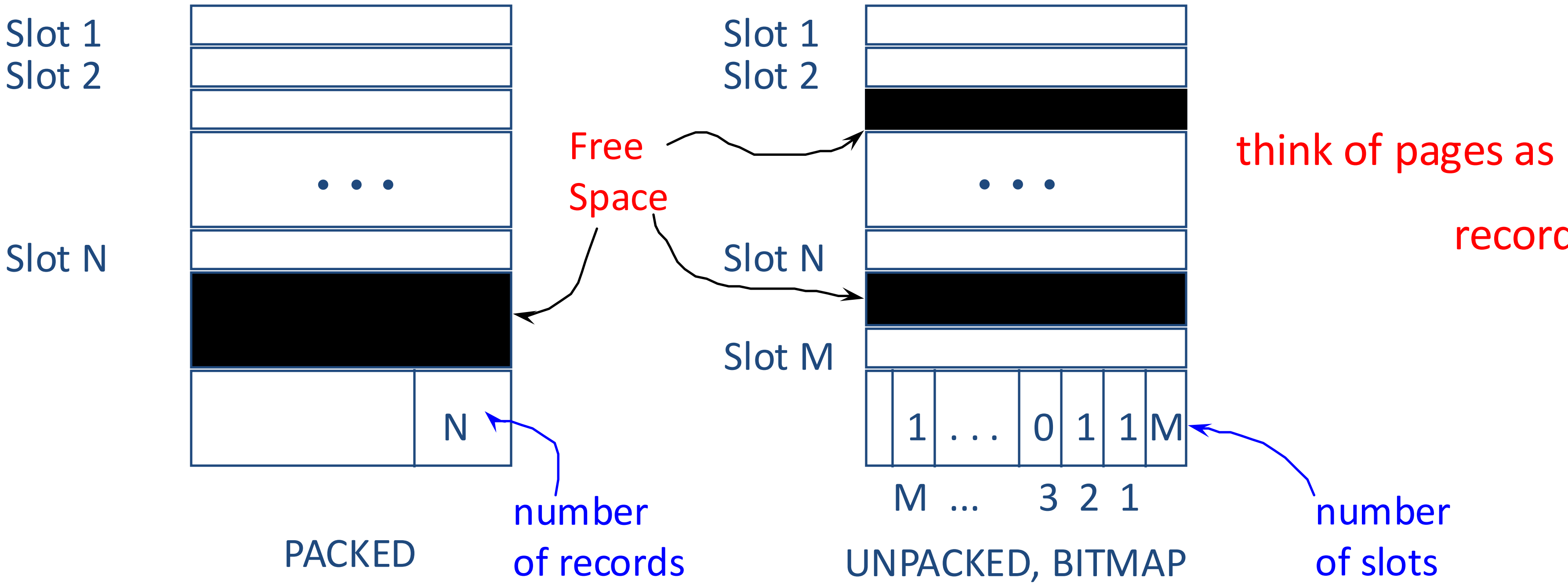


information about field types same for all records in a file; stored in *system catalogs*

how do we know these?

finding i^{th} field done via arithmetic

Page Formats: Fixed Length Records



think of pages as an array of bytes!

records too!

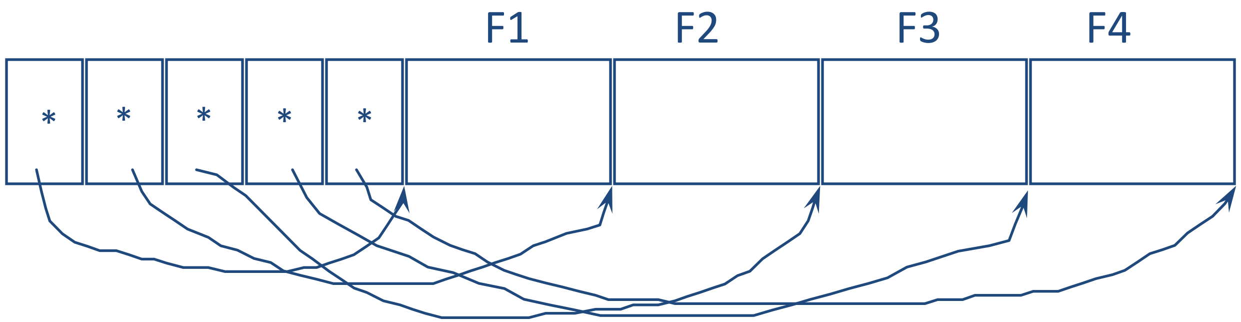
record id = <page id, slot #>

packed: moving records for free space management changes rid; may not be acceptable.

Variable Length is more complicated

Two alternative formats (# fields is fixed):

Fields Delimited by Special Symbols



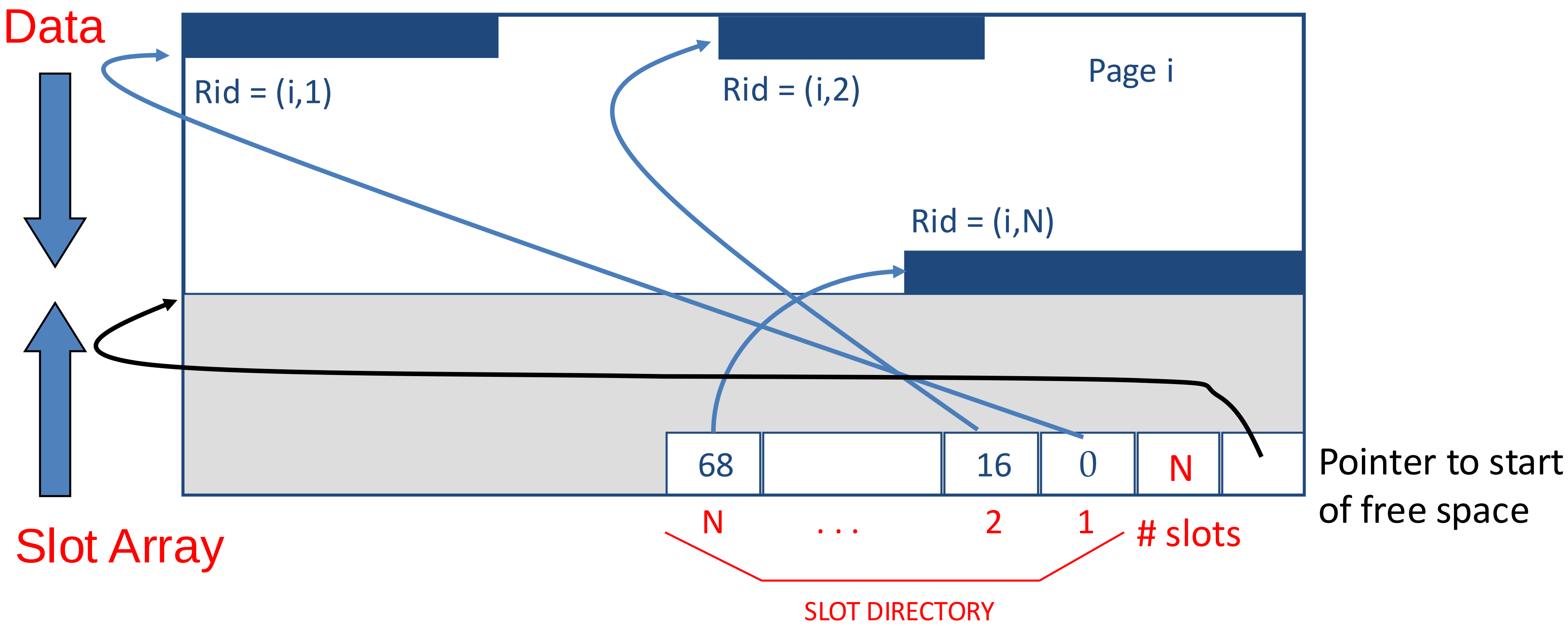
Array of Field Offsets

how?

Offset approach: **generally considered superior**

direct access to ith field and efficient storage of ***nulls***

“Slotted Page” for Variable Length Records

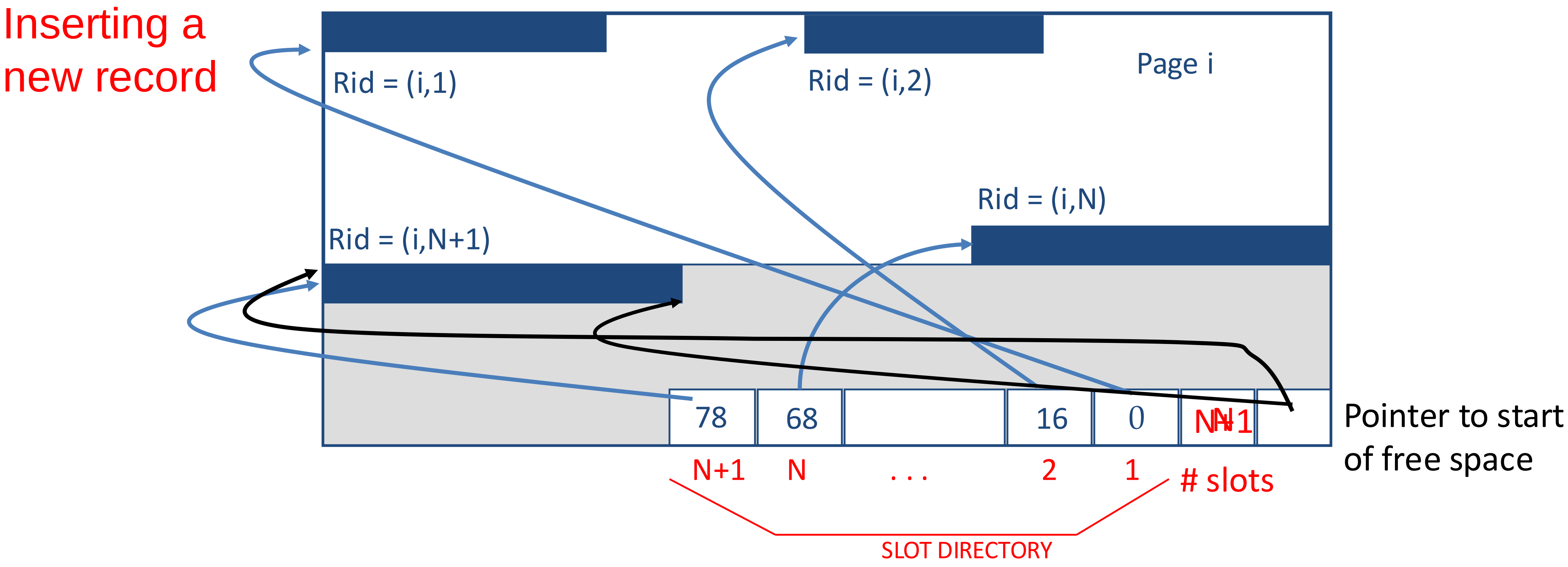


(1) *record id = <page id, slot #>*

(2) can move records on page without changing rid; so, attractive for fixed-length records too

(3) page is full when data space and slot array meet

"Slotted Page" for Variable Length Records



(1) *record id = <page id, slot #>*

(2) can move records on page without changing rid; so, attractive for fixed-length records too

(3) page is full when data space and slot array meet

Decomposition Storage Model (DSM)

R1

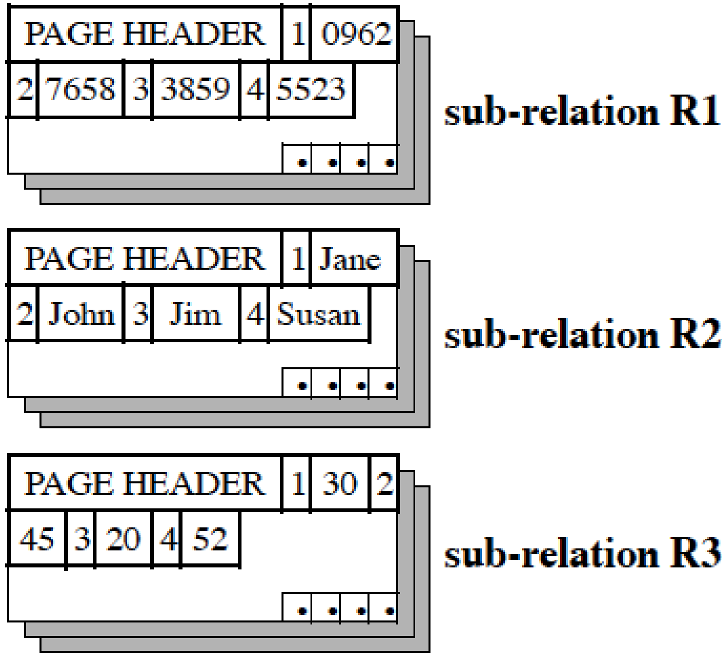
RID	SSN
1	0962
2	7658
3	3859
4	5523
5	9743
6	0618

R2

RID	Name
1	Jane
2	John
3	Jim
4	Susan
5	Leon
6	Dan

R3

RID	Age
1	30
2	45
3	20
4	52
5	43
6	37



Decompose a relational table to sub-tables per attribute

Why (and when) is this beneficial?

✓ Saves IO by bringing only the relevant attributes

Partition Attributes Across (PAX)

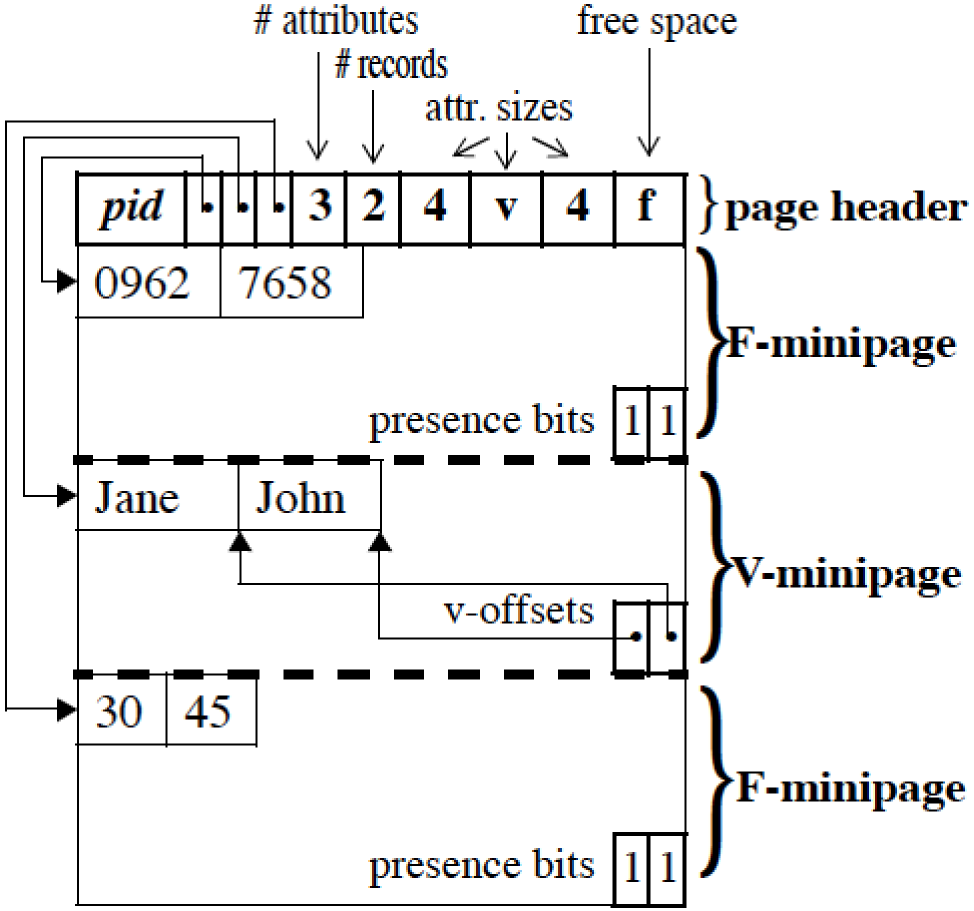
Middle ground?

Decompose a slotted-page internally in mini-pages per attribute

✓ Cache-friendly

- Brings only relevant attributes to cache

- Compatible with slotted-pages?
- Same update abstraction?
  - (insert in a page)



File Organization & Indexing

Page Layout (NSM, DSM, PAX)

File organization (Heap & sorted files)

Readings: Chapter 8.4 & 9.5

Index files & indexes

Index classification

Files

with traditional slotted pages

FILE:

A collection of pages, each containing a collection of records.

Must support:

- insert/delete/modify record
- read a particular record (specified using *record id*)
- scan all records (possibly with some conditions on the records to be retrieved)

Alternative File Organizations

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

- Heap files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best for retrieval in some order, or for retrieving a "range" of records.
- Index File Organizations: (will cover shortly..)

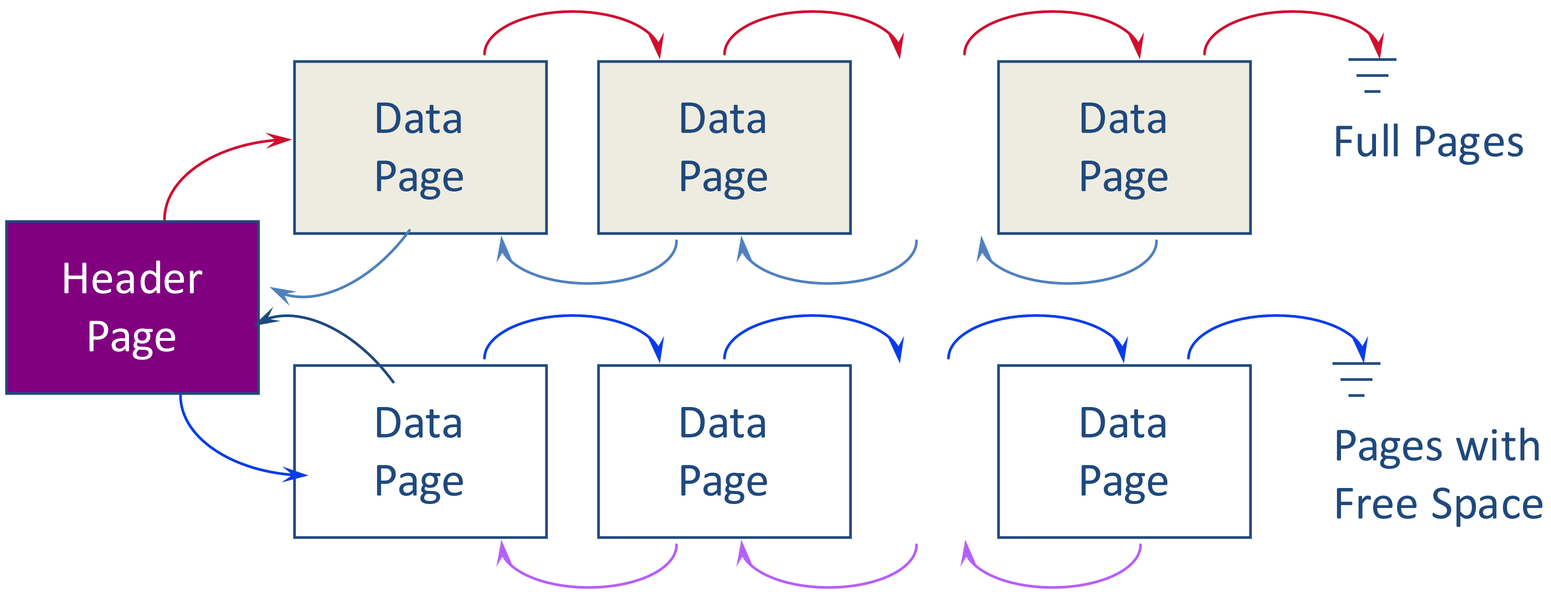
Heap (Unordered) Files

Simplest **file** structure

contains **records** in no particular order

As file grows and shrinks, disk pages are allocated / de-allocated

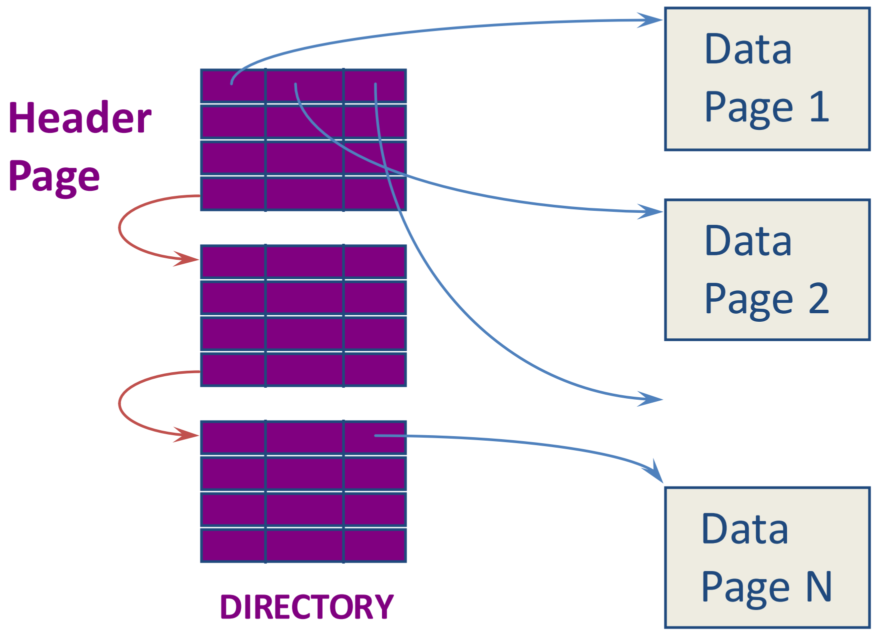
Heap File Implemented Using Lists



the header page id and Heap file name must be stored someplace

each page contains 2 “pointers” plus data

Heap File Using a Page Directory



The entry for a page can include the number of free bytes on the page.

The directory is a collection of pages; linked list implementation is just one alternative.

Much smaller than linked list of all HF pages!

Heap files vs. Sorted files

Quick+dirty cost model: # of disk I/O’s

For simplicity, ignore:

- CPU costs
- Gains from pre-fetching and sequential access

Average-case analysis; based on several simplistic assumptions.

Good enough to show the overall trends!

Some Assumptions in the Analysis

Single record insert and delete.

Equality search - exactly one match (e.g., search on key)

- Question: what if more or fewer???

Heap Files:

- Insert always *appends* to end of file.

Sorted Files:

- Files *compacted* after deletions.
- Search done on file-ordering attribute.

Cost of Operations (in #of I/Os)

B: Number of data pages

	Heap File	**Sorted File**	**notes…**
Scan all records	B	B	
Equality Search	0.5B	$\log_2 B$	*assumes exactly one match!*
Range Search	B	$(\log_2 B)$ + (#match pages)	
Insert	2	$(\log_2 B) + 2*(B/2)$	*must R & W*
Delete	0.5B + 1	$(\log_2 B) + 2*(B/2)$	*must R & W*

System Catalogs

For each relation:

name, file name, file structure (e.g., Heap file)

attribute name and type, for each attribute

index name, for each index

integrity constraints

For each index:

structure (e.g., B+ tree) and search key fields

For each view:

view name and definition

Plus stats, authorization, buffer pool size, etc.

Catalogs are themselves stored as relations!

Attr_Cat(attr_name, rel_name, type, position)

Candidate keys?

attr_name	rel_name	type	position
attr_name	Attr_Cat	string	1
rel_name	Attr_Cat	string	2
type	Attr_Cat	string	3
position	Attr_Cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

→ Try querying the PostgreSQL/MySQL catalogues (in SQL!)

File Organization & Indexing

Page Layout (NSM, DSM, PAX)

File organization (Heap & sorted files)

Index files & indexes

Readings: Chapter 8.2, 8.3.2

Index classification

Indexes

retrieve records searching *in one or more fields*:

Find all students in the “CS” department

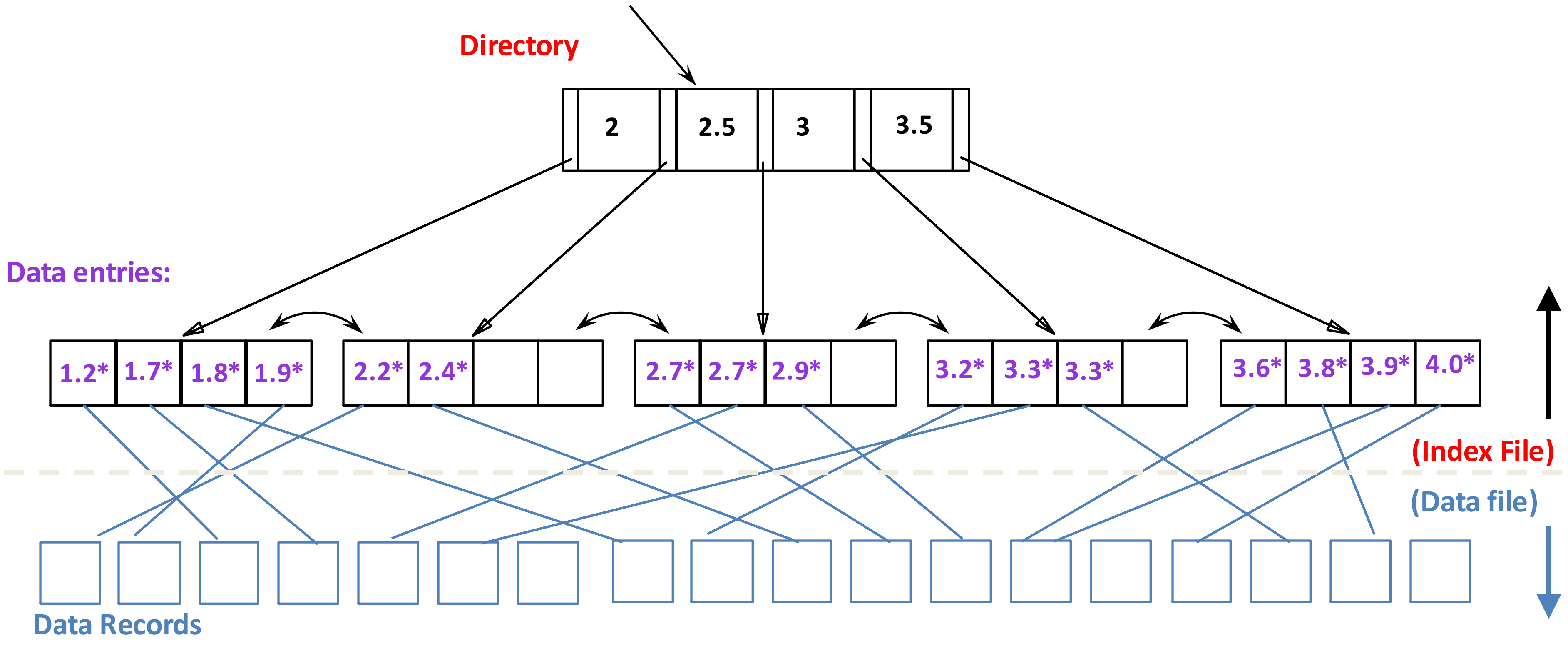
Find all students with a gpa > 3

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

search key is not the same as *key* (does **not** have to be unique).

Example: Simple Index on GPA



An index contains a collection of **data entries**, and supports efficient retrieval of **records** matching a given **search condition**

Index Search Conditions

Search condition = *<search key, comparison operator>*

Examples…

(1) Condition: Department = “CS”

- Search key: “CS”
- Comparison operator: equality (=)

(2) Condition: GPA > 3

- Search key: 3
- Comparison operator: greater-than (>)

File Organization & Indexing

Page Layout (NSM, DSM, PAX)

File organization (Heap & sorted files)

Index files & indexes

Index classification

Readings: Chapter 8.3, 8.5

Index Classification

Representation of data entries in index

i.e., what is at the bottom of the index? (3 alternatives)

Index Types

i. Clustered vs. Unclustered
ii. Primary vs. Secondary
iii. Dense vs. Sparse
iv. Single Key vs. Composite

Indexing technique

Tree-based, hash-based, other

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

1. Actual data record (with key value **k**)
2. <**k**, rid of matching data record>
3. <**k**, list of rids of matching data records>

Choice is **orthogonal** to the **indexing** technique.

- Examples of indexing techniques: B+ trees, hash indexes, R trees, …
- Typically, index contains auxiliary info that directs searches to the desired data entries

Can have **multiple** (different) **indexes** per file.

- E.g. file sorted on *age*, with a hash index on *name* and a B+tree index on *salary*.

Alternatives for Data Entries (Contd.)

Alternative 1:

Actual data record (with key value **k**)

k	record: <att1, att2, …>

- The index structure **is a file organization** for data records (like Heap Files or Sorted Files).
- **At most one index** per relation can use Alternative 1.
- It saves pointer lookups but can be expensive to maintain with insertions and deletions.

Alternatives for Data Entries (Contd.)

Alternative 2

<**k**, rid of matching data record>

k	record_id: <rid>

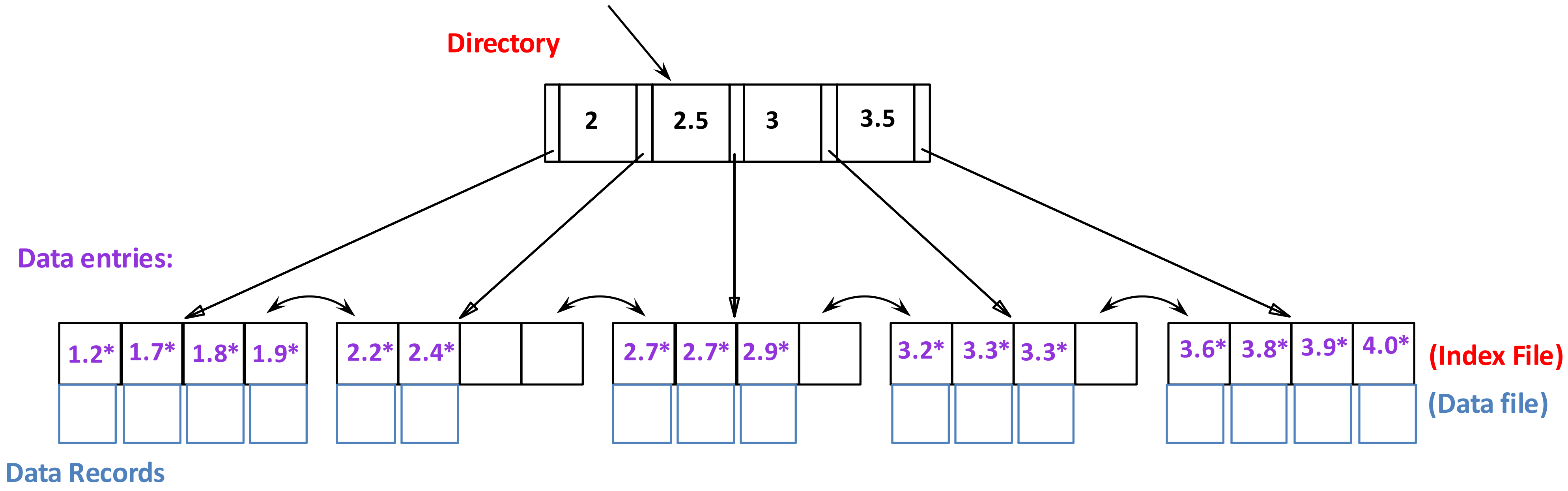
and Alternative 3

<**k**, list of rids of matching data records>

k	record_ids: <rid1, rid2, …>

- **Easier** to maintain than Alternative 1.
- **At most one index** can use Alternative 1; **rest must use** Alternatives **2 or 3**.
- **Alternative 3 more compact than Alternative 2**
  - but leads to *variable sized data entries* even if search keys are of fixed length.
- Even worse, for **large rid lists** the data entry would have to **span multiple pages**!

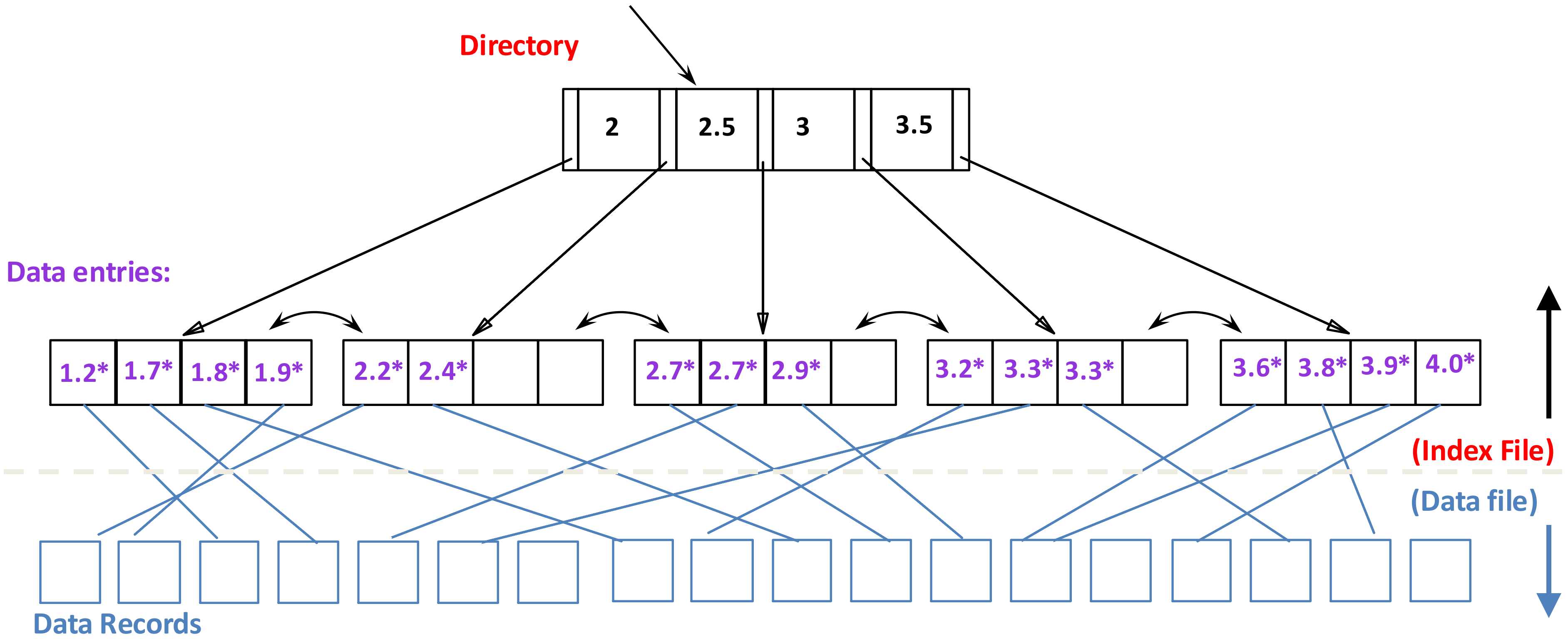
Alternatives for Data Entries



Alternative 1:

k	record: <att1, att2, …>

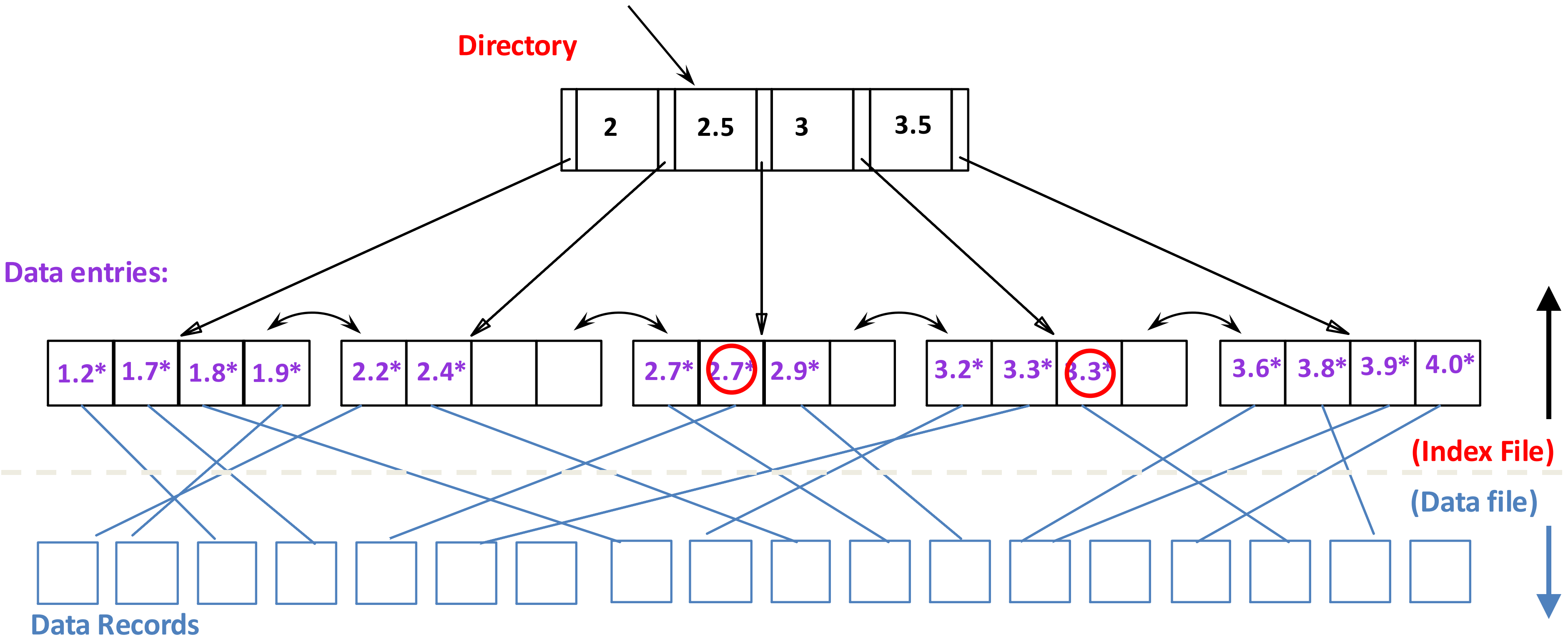
Alternatives for Data Entries



Alternative 2:

k	record_id: <rid>

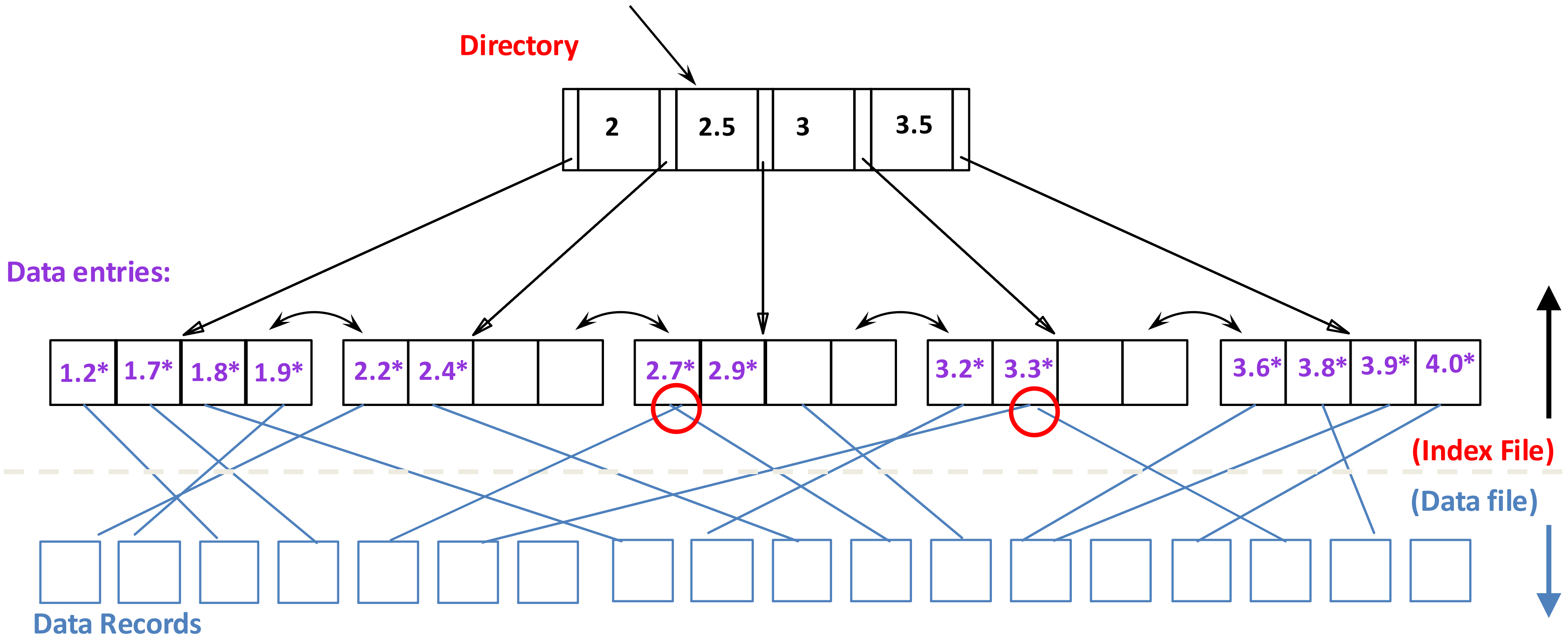
Alternatives for Data Entries



Alternative 3:

k	record_ids: <rid1, rid2, …>

Alternatives for Data Entries



Alternative 3:

k	record_ids: <rid1, rid2, …>

Where were we? Index Classification

Representation of data entries in index

i.e., what is at the bottom of the index? (3 alternatives)

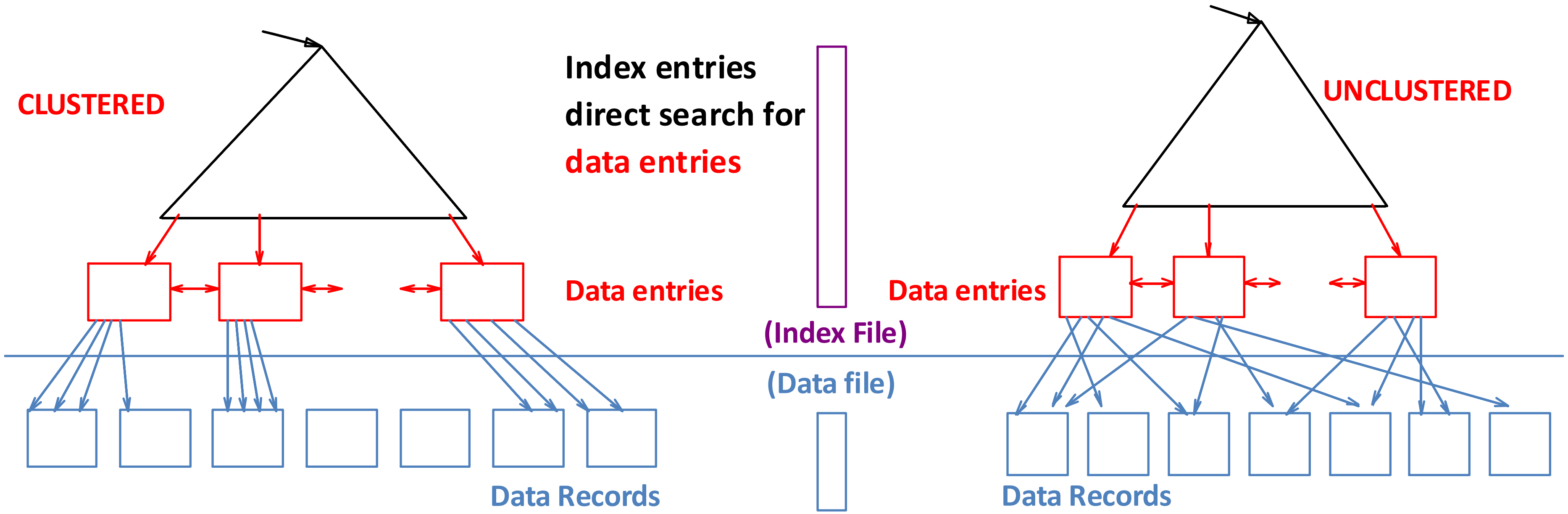
i. Clustered vs. Unclustered
ii. Primary vs. Secondary
iii. Dense vs. Sparse
iv. Single Key vs. Composite

Indexing technique

Tree-based, hash-based, other

Index Classification - clustering

Clustered vs. *unclustered*: If order of data records is the same as, or "close to", order of index data entries, then called *clustered index*.



Index Classification - clustering

A file can have a clustered index on *at most one* key.

is it always true?

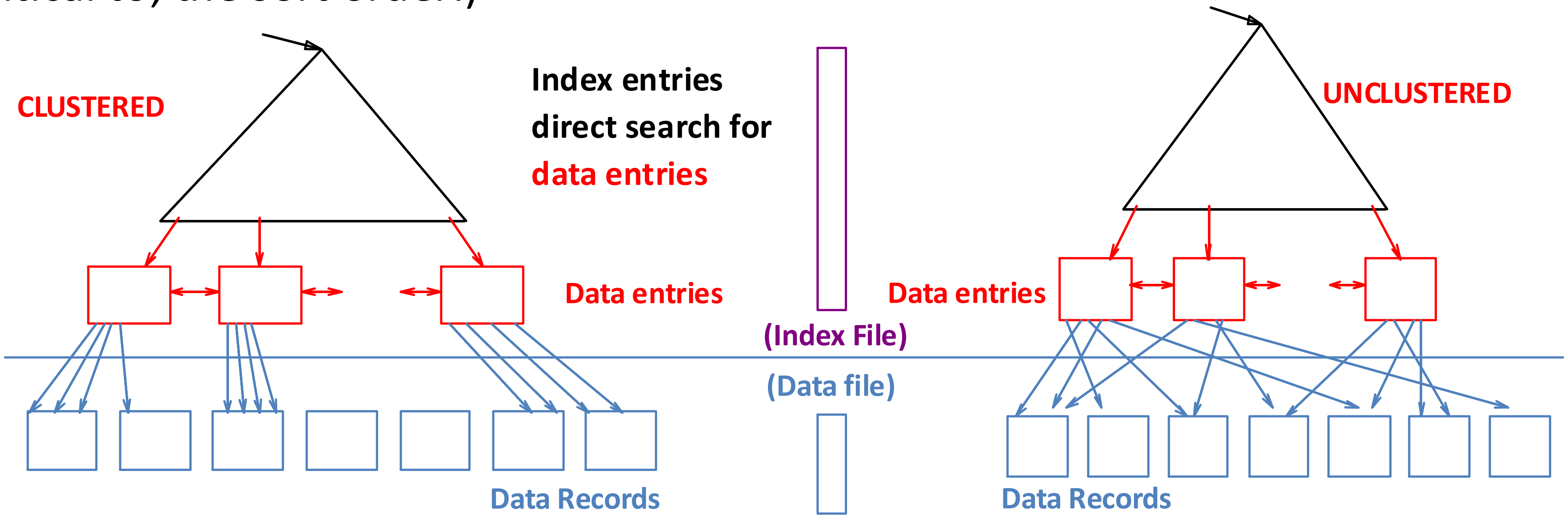
Cost of retrieving data records through index *varies greatly* based on whether index is clustered!

Note: Alternative 1 implies clustered, *but not vice-versa*.

Index Classification - clustering

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.)



Index Classification - clustering

Cost of retrieving records found in range scan:

Clustered: cost = # pages in file w/matching records

Unclustered: cost ≈ # of matching index <u>data entries</u>

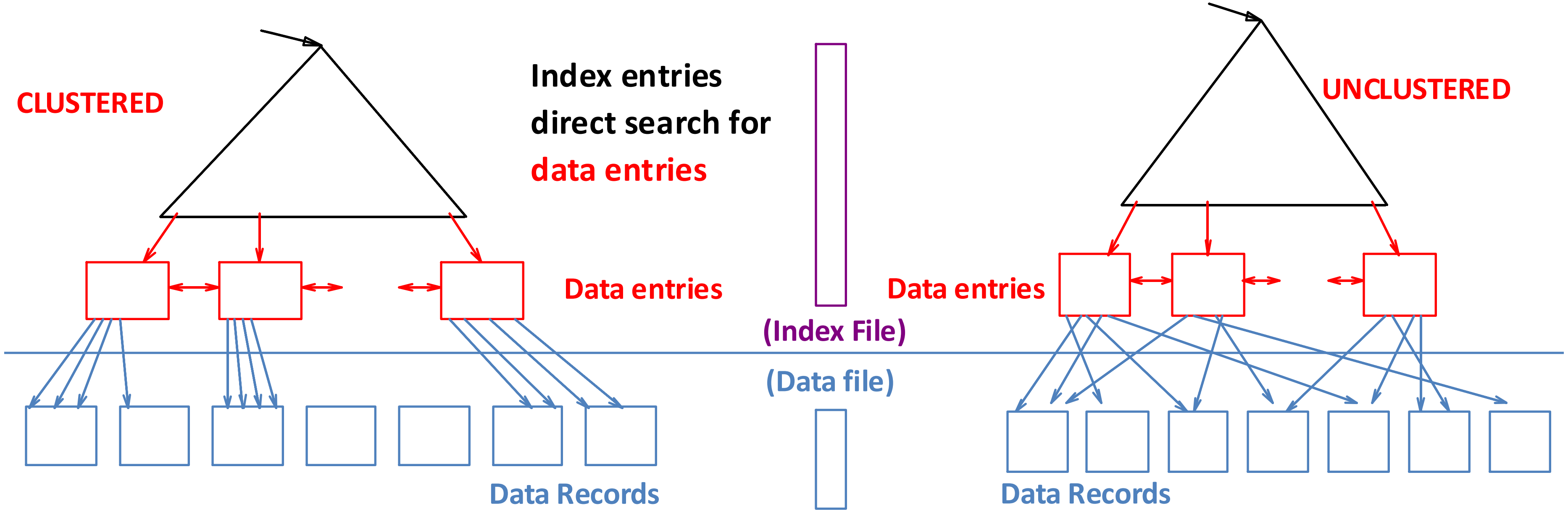
What are the tradeoffs????

Clustered Pros:

- ✓ <u>Efficient range searches</u>
- ✓ Compression

Clustered Cons:

- x <u>expensive to maintain</u>
  - on-the-fly
  - sloppy with re-org



Primary vs. Secondary Index

Primary: index key includes the file's primary key

Secondary: any other index

Sometimes confused with Alt. 1 vs. Alt. 2/3

Primary index *never contains duplicates*

Secondary index *may contain duplicates*

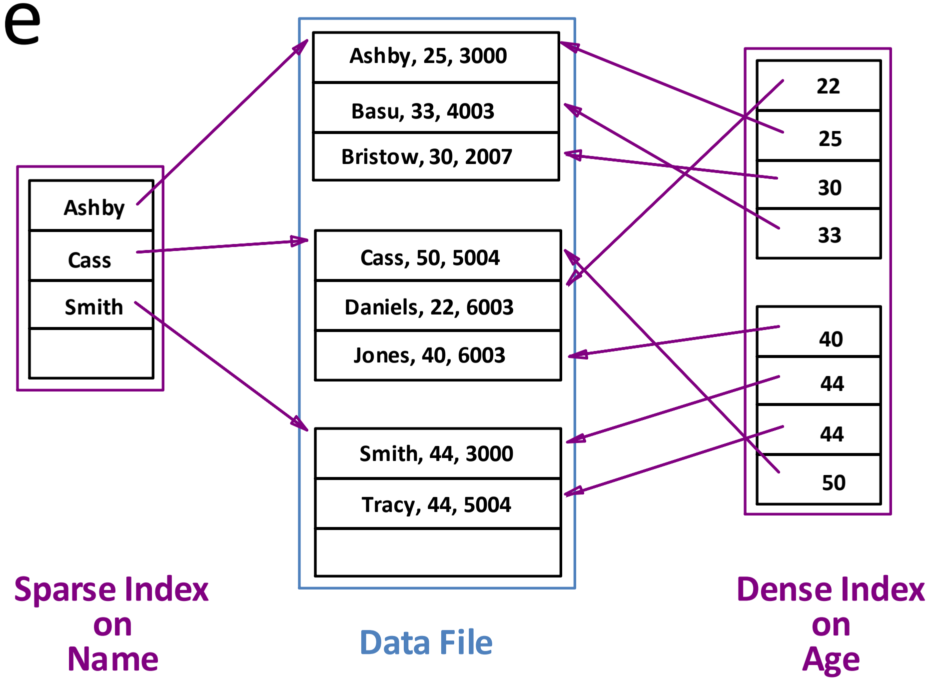
If index key contains a candidate key, no duplicates => unique index

Dense vs. Sparse Index

Dense: at least one data entry per key value

Sparse: an entry per data page in file

- Every **sparse index is clustered**!
- **Sparse** indexes are **smaller**;
  - however, some useful
  - optimizations are based
  - on dense indexes.
- **Alternative 1** always leads to **dense** index.



Composite Search Keys

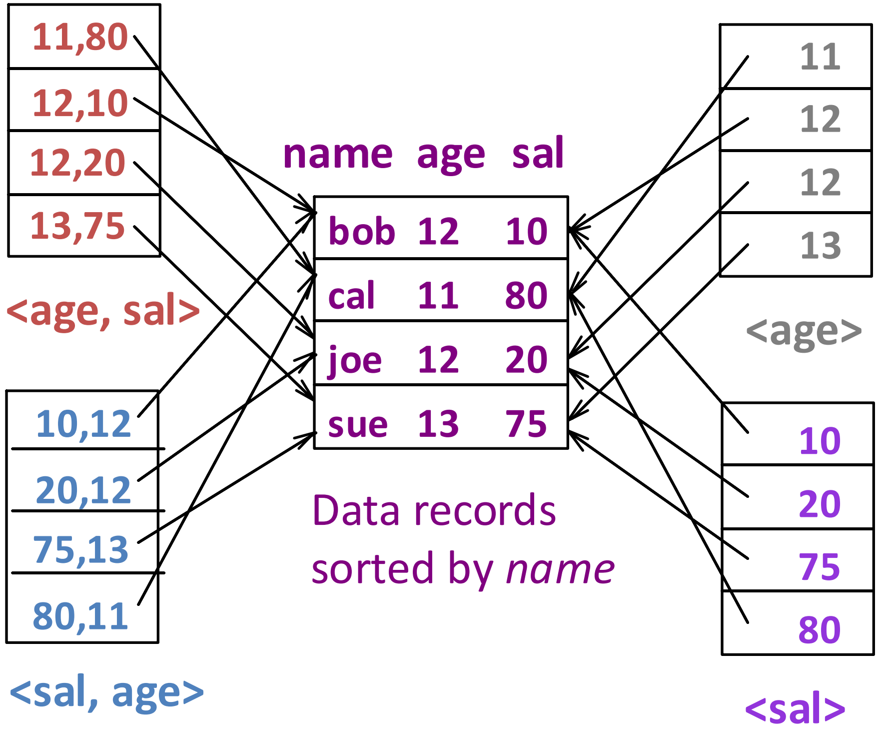
Search on *combination* of fields.

- Equality query: Every field is equal to a constant value. E.g. wrt <sal,age> index:
  - age=12 and sal =75
- Range query: Some field value is not a constant, e.g.:
  - age=12; or age=12 and sal > 20; or age>15

Data entries in index sorted by search key for range queries

- “Lexicographic” order

Examples of composite key indexes using lexicographic order.



Tree vs. Hash-based index

Hash-based index: Good for equality selections.

File = a collection of *buckets*. Bucket = *primary* page plus 0 or more *overflow* pages

Hash function **h**: **h**(*r.search_key*) = bucket in which record *r* belongs

Tree-based index: Good for range selections.

Hierarchical structure (Tree) directs searches

Leaves contain data entries sorted by search key value

B+ tree: all root->leaf paths have equal length *(height)*

Will discuss in Classes 7 & 8!
More indexes in Classes 11 & 12!

Summary

variable length record format

with field offset directory supports direct access to i^{th} field and null values

slotted page

supports variable length records and allows records to move on page

file layer

(i) keeps track of pages in a file (ii) supports abstraction of a collection of records
(iii) *tracks availability of free space*

catalog relations

store metadata about relations, indexes, views (*common to all records in a given collection*)

Summary (Cont.)

various file organizations

sorting or building an *index* is important if selection queries are frequent

index is a collection of data entries plus a way to quickly find entries with given key values.

(i) hash-based good for equality search

(ii) sorted files and tree-based indexes best for range search; also good for equality search

(files not kept sorted in practice; B+ tree more common)

Summary (Cont.)

data entries in index can be: (i) actual data records, (ii) <key, rid> pairs, or (iii) <key, rid-list> pairs.

[orthogonal to *indexing structure (i.e. tree, hash, etc.)*]

usually have several indexes on a given file of data records, each with a different search key

index classification

(i) clustered vs. unclustered, (ii) primary vs. secondary, (iii) sparse vs. dense, (iv) single key vs. composite key

affect utility & performance