CS122A: Introduction to Data Management

Lecture #14: Indexing

Instructor: Chen Li

Indexing in MySQL (w/InnoDB)

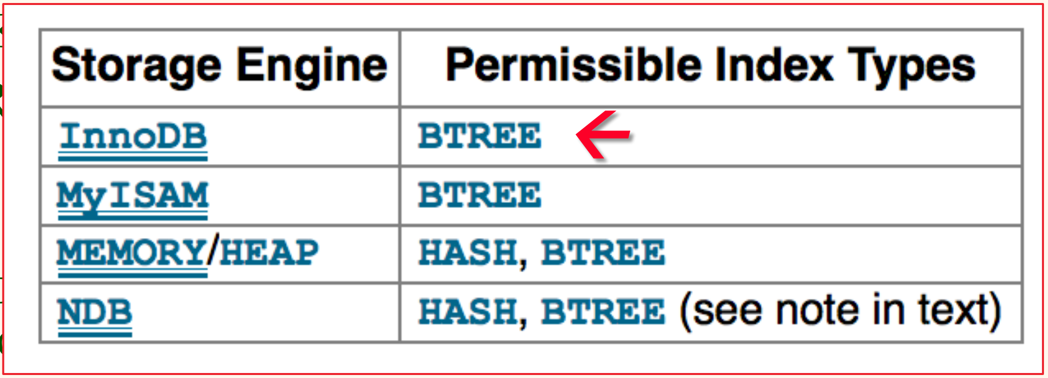
CREATE [UNIQUE | FULLTEXT | SPATIAL**] INDEX** *index_name*
[index_type] ON *tbl_name* **(index_col_name,...) [index_option]**
[algorithm_option | lock_option] ...

index_col_name: col_name [(length)] [ASC | DESC]

index_option: index_type | K
| WITH PARSER

index_type: USING {BTREE

algorithm_option: ALGORITHM [=] {D
lock_option: LOCK [=] {DEFAULT | N



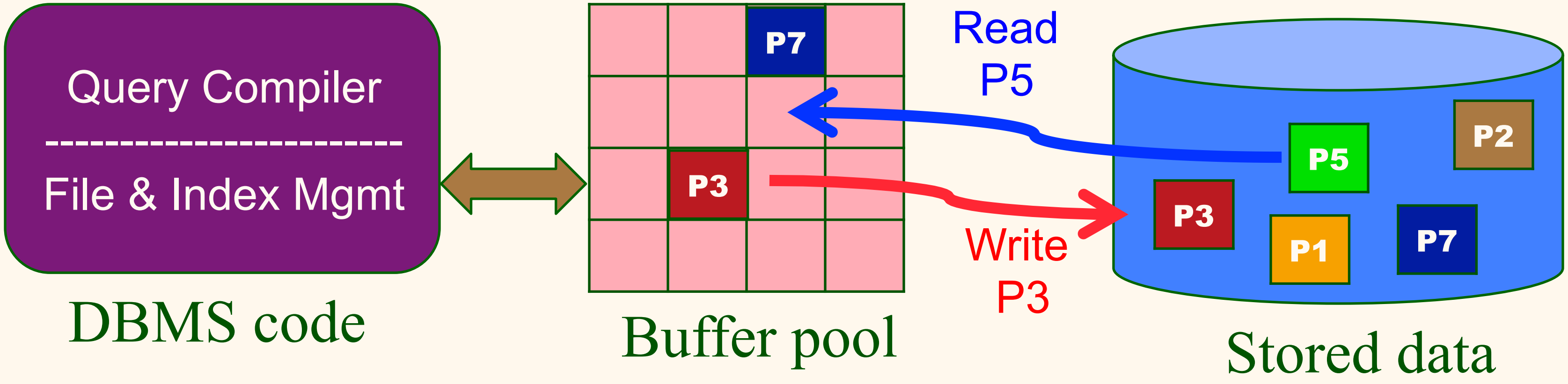
Storage Engine	Permissible Index Types
InnoDB	BTREE ←
MyISAM	BTREE
MEMORY/HEAP	HASH, BTREE
NDB	HASH, BTREE (see note in text)

Ex: **CREATE INDEX** salidx **ON** Emp (sal) **USING BTREE;**

(http://dev.mysql.com/doc/refman/5.7/en/create-index.html)

Disks and Files

- DBMSs store information on ("hard") disks.
- This has major implications for DBMS design!
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be considered carefully!

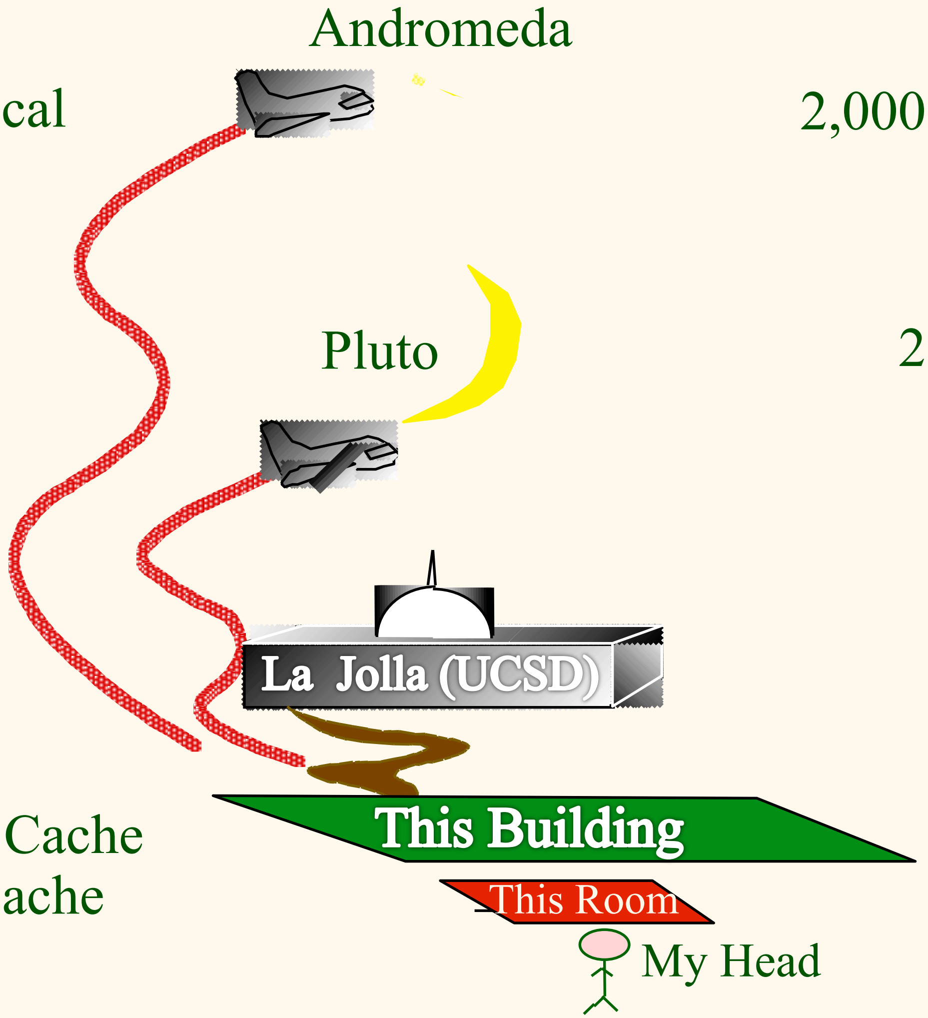


DBMS code

Buffer pool

Stored data

Storage Hierarchy & Latency (Jim Gray): How Far Away is the Data?

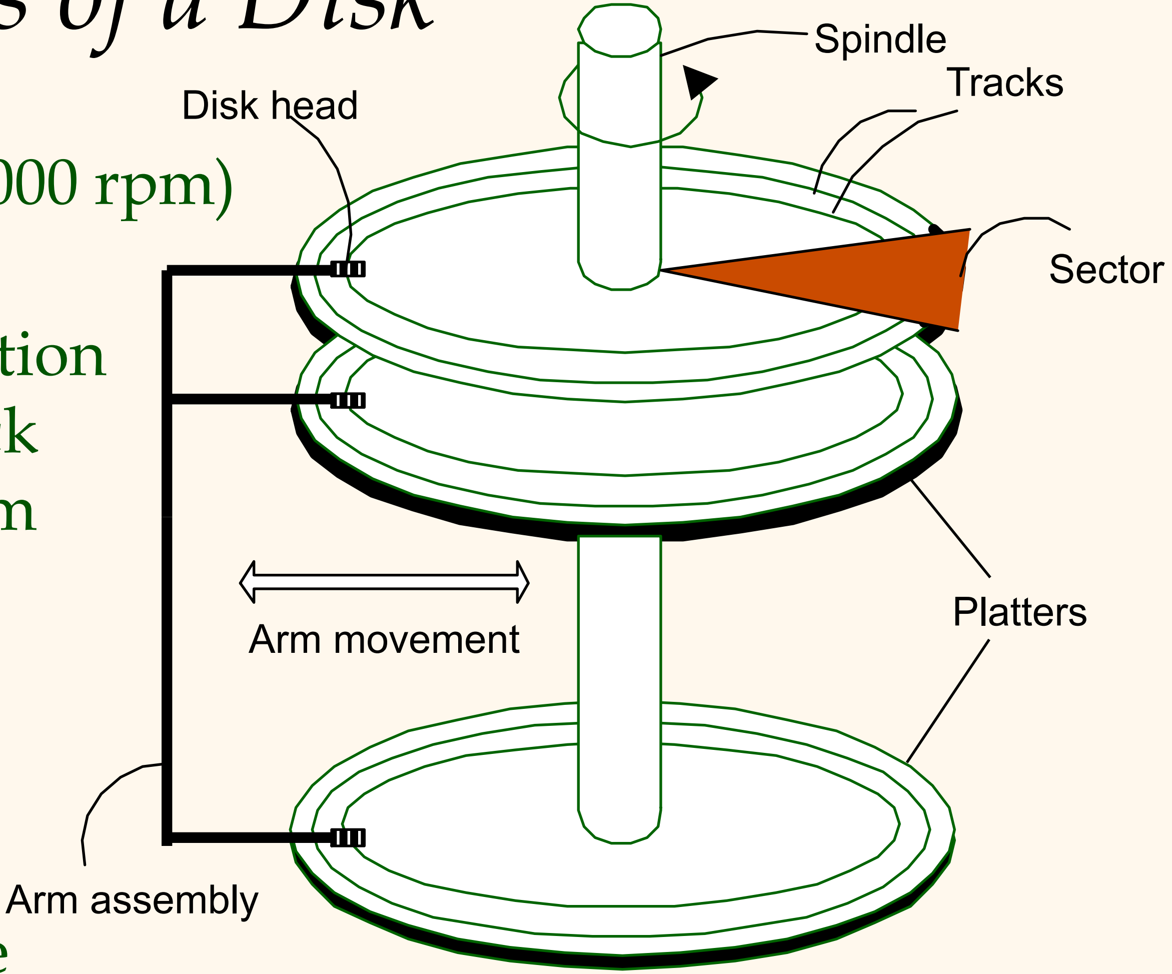


10^9	Tape /Optical Robot	Andromeda	2,000 years
10^6	Disk	Pluto	2 years
100	Memory	La Jolla (UCSD)	1.5 hr
10	On Board Cache	This Building	10 min
2	On Chip Cache	This Room	
1	Registers	My Head	1 min

Why Not Store Data in Main Memory??

- *Costs too much.* Dell was recently asking $65 for 500GB of disk, $600 for 256GB of SSD, and $57 for 4GB of RAM → *$0.13, $2.34, $14.25 per GB*
- *Main memory is volatile.* We want data to be saved between runs. (Obviously!!)
- Your typical (basic) storage hierarchy:
  - Main memory (RAM) for currently used data
  - Disk for the main database (secondary storage)
  - Tapes for archiving the data (tertiary storage)

Components of a Disk



- The platters spin (10,000 rpm)
- The arm assembly is moved in or out to position a head on a desired track Tracks under heads form a *cylinder* (imaginary!)
- Only one head reads/writes at any one time.
- *Block size* is a multiple of *sector size* (which is fixed)

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 < 1 msec per 4KB page
  - Key to lowering I/O cost: Reduce seek/rotation delays! → Bottom line: *Random vs. sequential I/O*

Ex: Emp(<u>eid,</u> ename, sal, deptid)

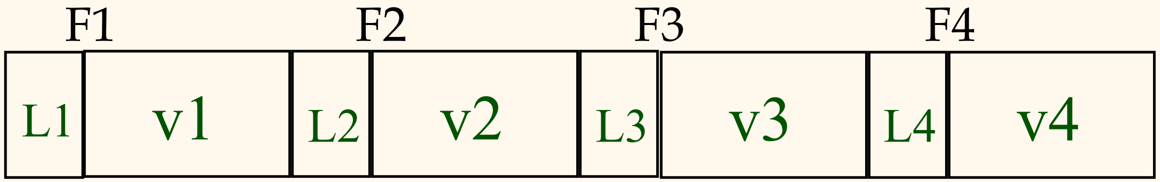
P1	1	111	Smith	3K	1
	2	222	Lee	100K	3
	3	333	Carey	80K	1
	4	444	Smith	12K	7
P2	1	555	Smith	18K	3
	2	666	Jones	90K	5
	3	777	Smith	23K	4
	4	888	Krishan	60K	8
		…			
P10000	1	…	…	…	…
	2	…	…	…	…
	3	…	…	…	…
	4	9999999	Smith	18K	11

Underlying *Emp* file pages

← Record id (RID) is (P2,3)

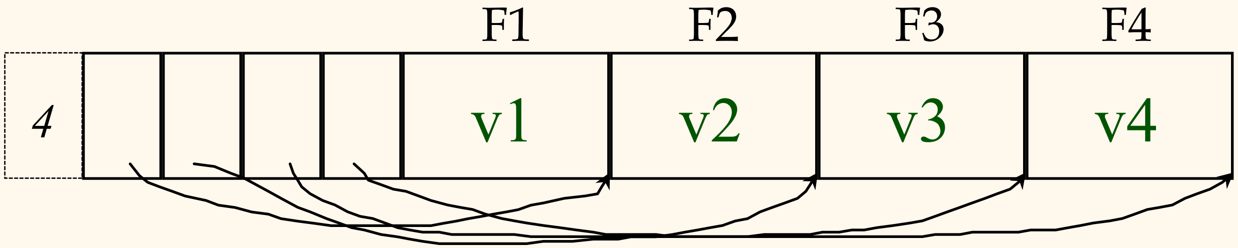
Record Formats: Variable Length

- Several alternative formats (# fields is fixed):



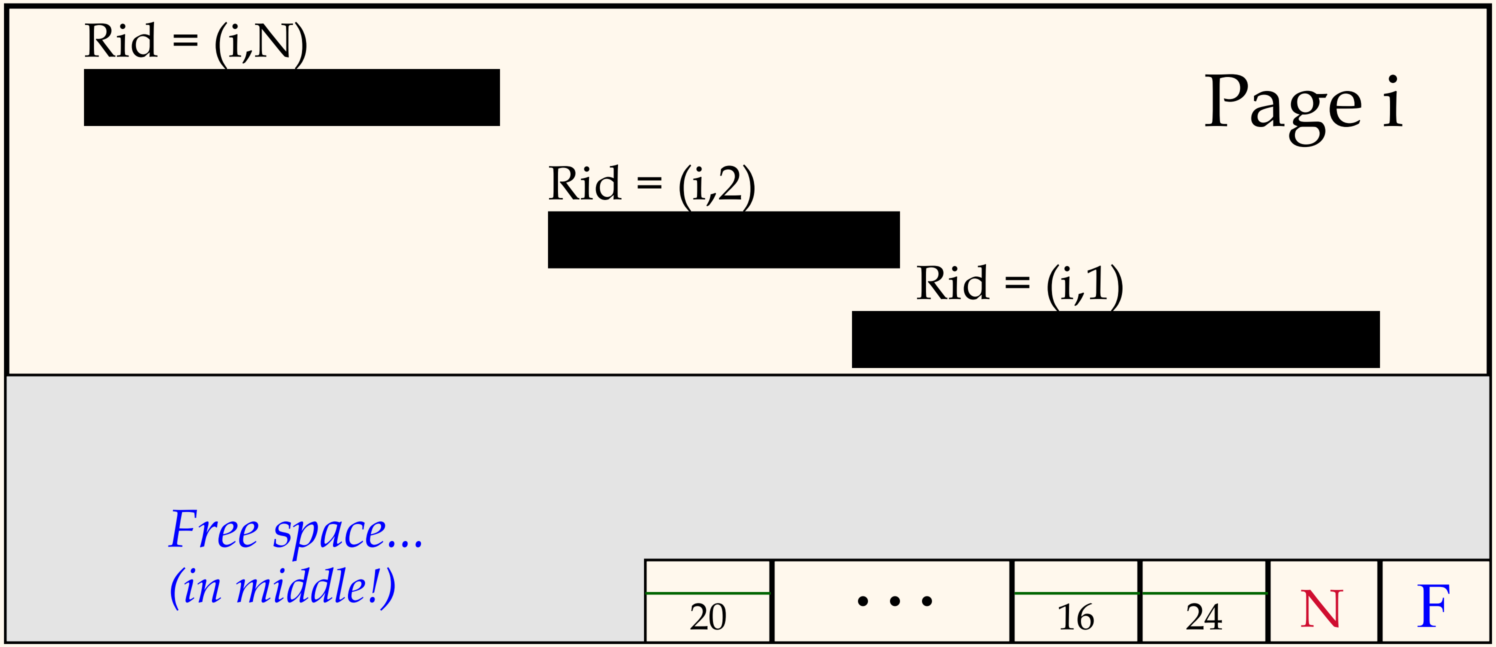
Fields Preceded by Field Lengths

- Variable-length fields with a directory:



Array of field offsets (*a.k.a.* directory)

Page Formats: Variable Length Records



Rid = (i,N)

Page i

Rid = (i,2)

Rid = (i,1)

Free space...
(in middle!)

20	• • •	16	24	N	F

SLOT DIRECTORY (offset, length)

- *Can move records within page w/o changing RIDs; not so unattractive for fixed-length records as a result.*

Processing a Query

- Suppose someone asks a simple SQL query:
  - SELECT * FROM Emp WHERE eid = 12345;
- Processing options include:
  - *Option 1*: Sequentially scan the data file (and stop, if we know eid is a key) → 5000 page reads (avg.)
  - *Option 2*: Binary search the data file (and stop, if we know eid is a key) → $\log_2(10,000) \approx 15$ page reads (avg.)
  - Even though Option 2 is ≈ 30x faster, we'd like to do even better (especially) for large data sets!!)

Indexing is the Answer!

Key k or key range (k_1, k_2)

888

Index on k

I(k) *or* I(k_1), ..., I(k_2)

(P2,4)

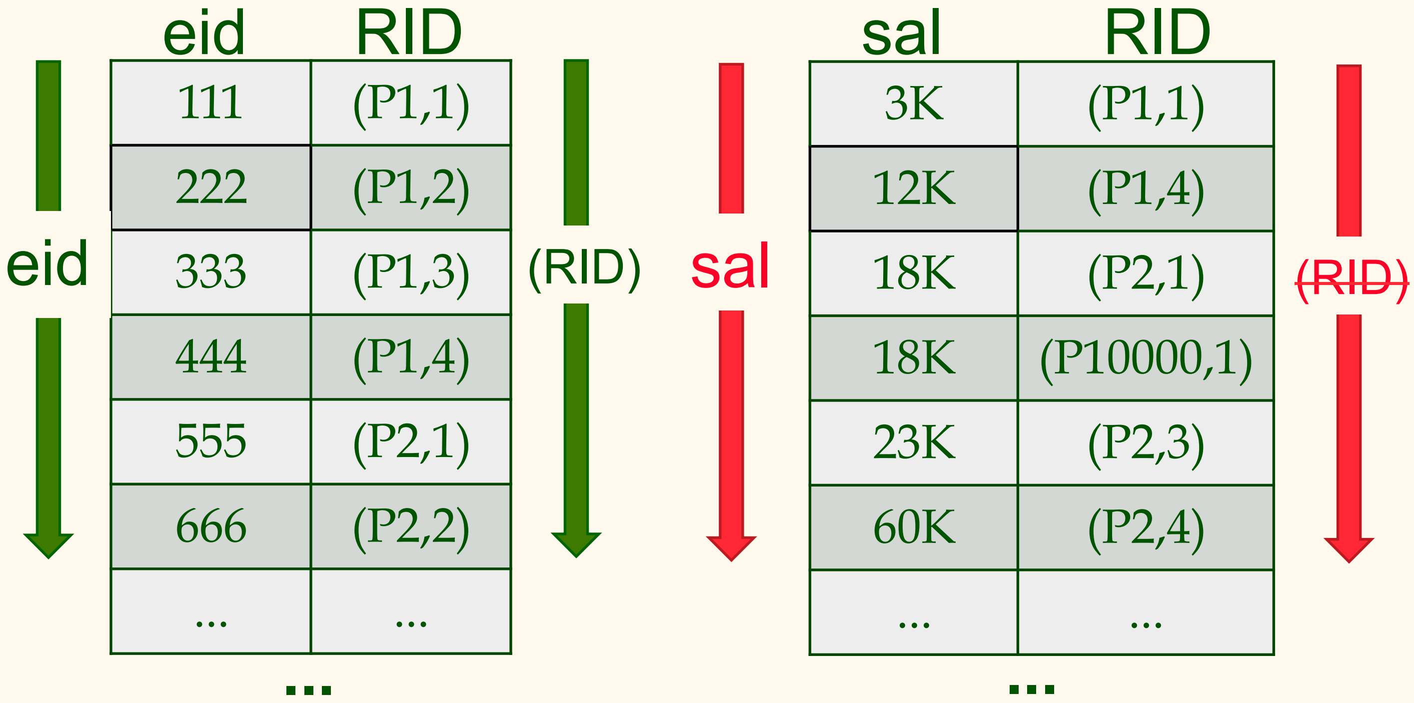
- Index maps from keys to associated info I
  - I(k) can be the *data record* with key k, or
  - I(k) can be the *RID* of the data record with key k, or
  - I(k) can be a *list of RIDs* of data records with key k!
  - Alternatively, we could map from data field values to the *PK value(s)* of the associated record(s)

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 serve as the search key for an index on the relation.
  - *Search key* is **not** the same as a *key* (i.e., it's not the primary key, it's a field we're very interested in).
- An index contains a collection of *data entries*, and it supports efficient retrieval of *all* data entries ***k**** with a given key value *k*.
  - Given a data entry *k**, we can find 1st record with key *k* with just more disk I/O. (Details soon …)

Ex: Emp(<u>eid</u>, ename, sal, deptid)

- One simple approach would be to have another file, sorted *on k*, for each *k* that we want to index
  - Hundreds of (key, RID) entries will fit on a single page
  - Index is thus much smaller than the data file
  - Less data (fewer reads) to search to locate the RIDs of interest



eid	RID
111	(P1,1)
222	(P1,2)
333	(P1,3)
444	(P1,4)
555	(P2,1)
666	(P2,2)
…	…

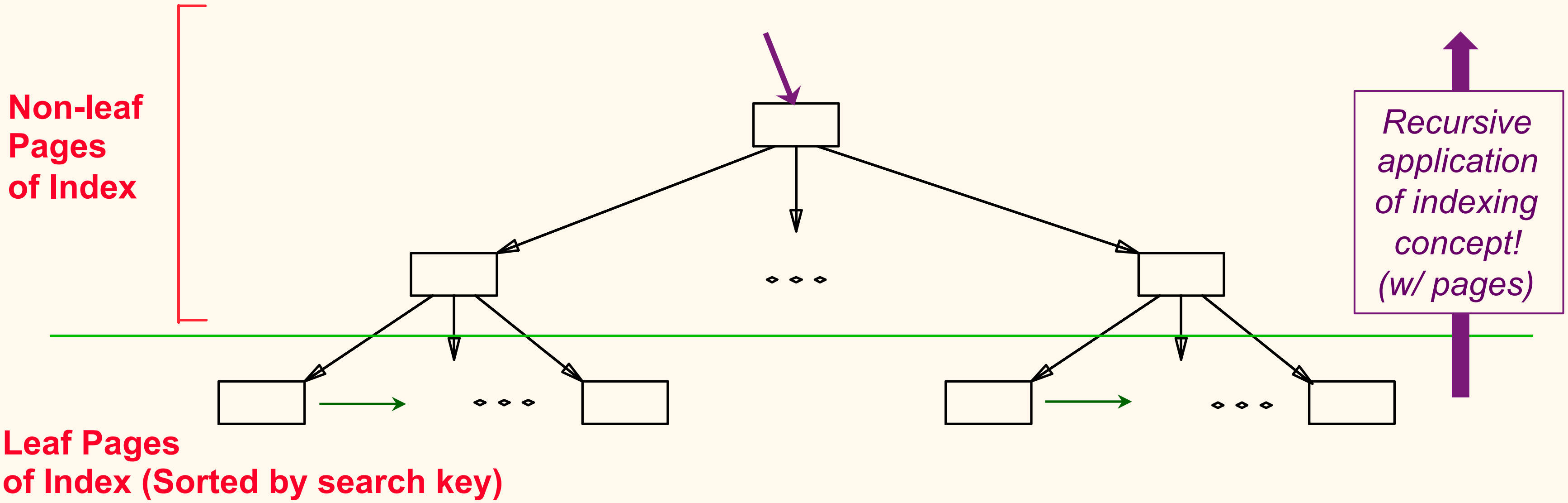
…

sal	RID
3K	(P1,1)
12K	(P1,4)
18K	(P2,1)
18K	(P10000,1)
23K	(P2,3)
60K	(P2,4)
…	…

…

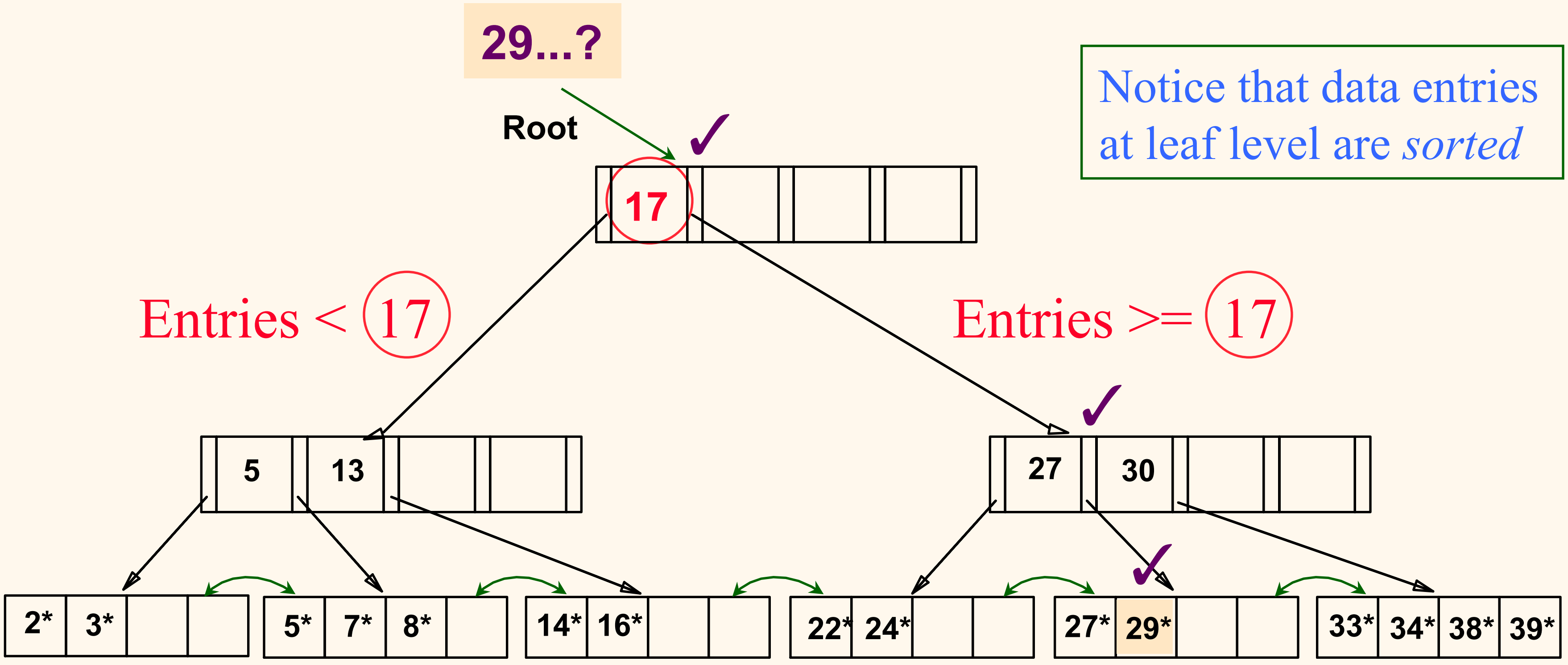
<u>Note:</u>
Can have multiple of these!

Even Better: Tree Indexes!



- Leaf pages contain *data entries,* and are chained
- Non-leaf pages have *index entries;* role is to guide searches
- Query processing steps become:
  1. Choose a good index to use (if one is available)
  2. Search the index to determine the interesting RID(s)
  3. Use the RID(s) to fetch the corresponding record(s)

An Example (B+ Tree)



29...?

Root

Entries < 17

Entries >= 17

Notice that data entries at leaf level are *sorted*

$k^* = (k, \mathrm{I}(k))$

__Note:__ Just *3 page reads* to get from root to (any) leaf here!

Index Classification

- *Primary* vs. *secondary*: If search key contains the primary key, then called the 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', the order of stored data records, then called a clustered index.
  - A table can be clustered on at most one search key.
  - Cost of retrieving data records via an index varies *greatly* based on whether index is clustered or not!

Clustered vs. Unclustered Indexes