Storage and Access

Columnar Storage

- Traditional: storing rows at a time
- Less updates, read-heavy, wide tables
  - storing a few, even 1, column(s) at a time
- why?
  - less I/O: only read the needed columns
  - better compression: uniformity of data type
  - better compression: run length encoding
  - delay assembling rows back

Columnar Storage

- Ultimate decomposition with a surrogate key!
- Not great for enforcing integrity constraints
- Not great for update performance
  - hybrid solutions can help

Basic Columnar Layout

Row Store

id	C1	C2	C3	C4
1				
2				
3				
4				
5				
6				

Column Store with explicit Ids

id	C1
1	
2	
3	
4	
5	
6	

id	C2
1	
2	
3	
4	
5	
6	

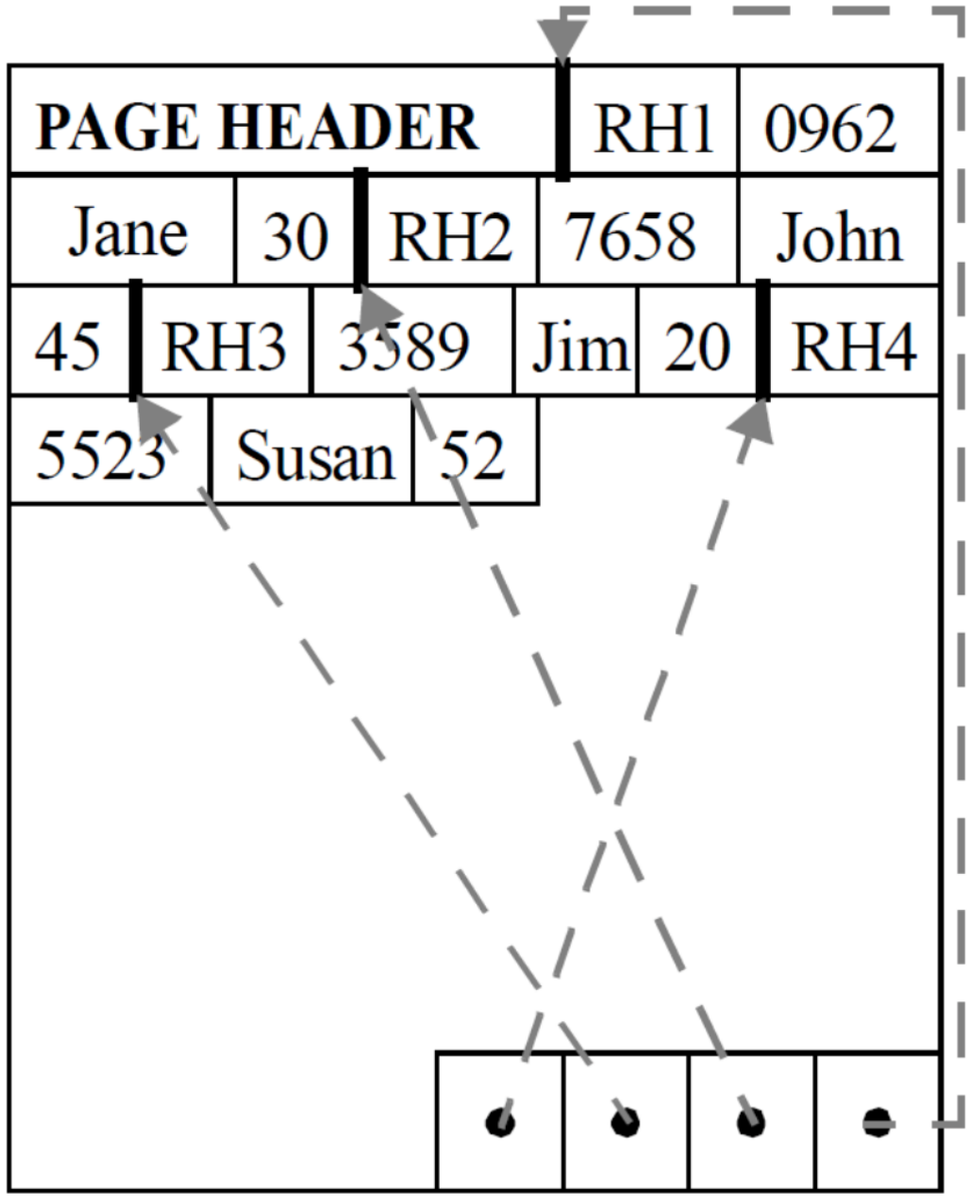
id	C3
1	
2	
3	
4	
5	
6	

id	C4
1	
2	
3	
4	
5	
6	

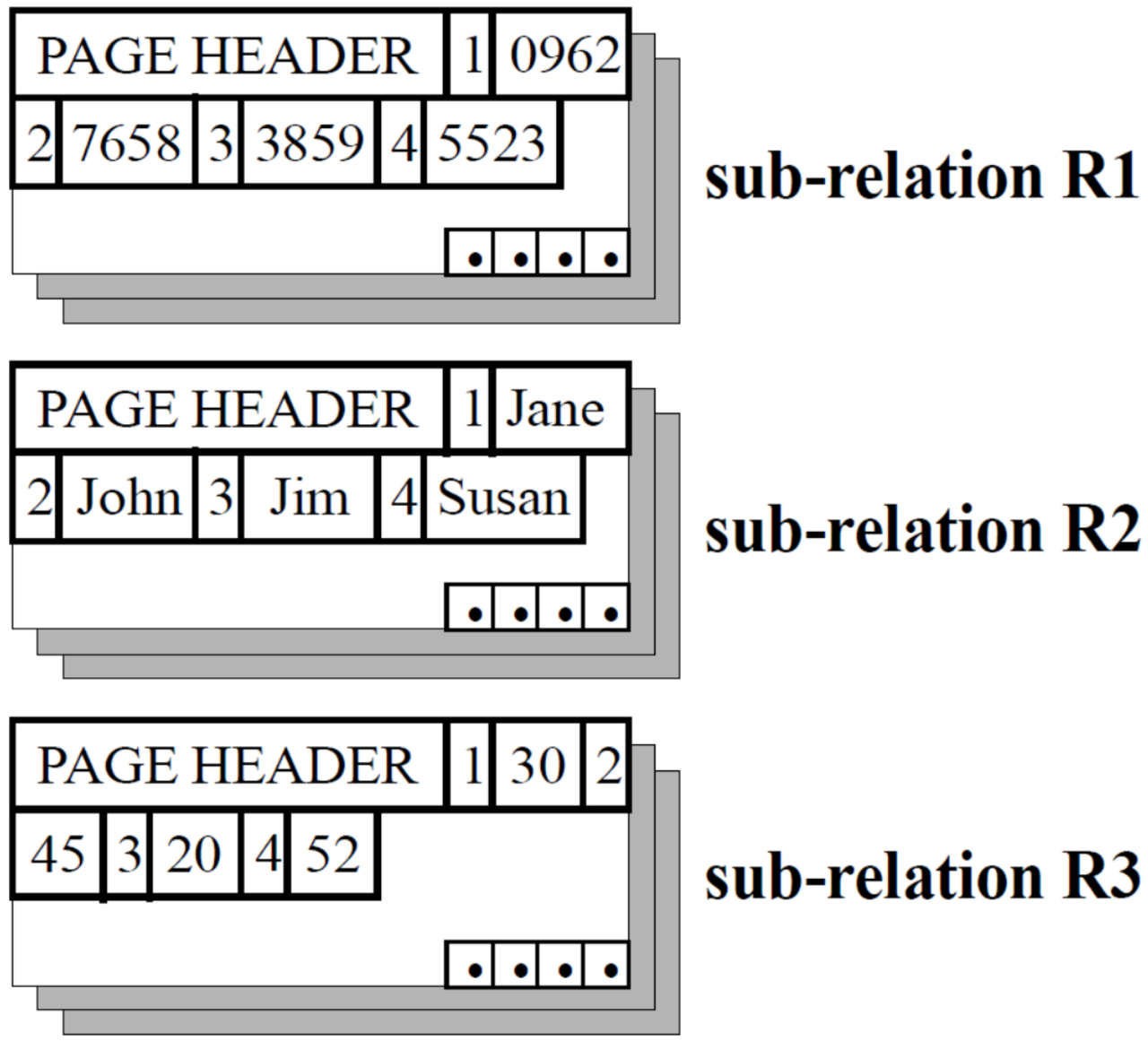
Column Store with virtual Ids

id	C1	C2	C3	C4
1				
2				
3				
4				
5				
6				

Inside the Page



NSM Page



DSM Pages

Columnar Summary

- Another tool in the database tool-kit
- Soon all DBMS vendors will have it

Data Compression Review

- Semantics preserving
  - Dictionary encoding: encode common values and decode using a dictionary
  - Run length encoding: repeated consecutive values are kept once with a count for the “repeats”
- Non-preserving
  - using standard compression techniques, e.g. LZW

Dictionary Encoding

- Let's say states NY, CA, TX, FL occur much more than others (use 2 bits to encode, all 4) then store the 2 bits instead of value.
- At read time, look up the value
- create table t1(…
  state char(2) compress ('NY', 'CA', 'TX', 'FL'), …)
- the dictionary must be fairly small (fit in memory)

Run Length Encoding

dept	cid
CS	367
CS	367
CS	367
CS	564
CS	564
CS	564
Math	201
Math	234
PolSci	104
PolSci	104
PolSci	104
PolSci	104

Run Length Encoded Table

dept	cid	count
CS	367	3
CS	564	3
Math	201	1
Math	234	1
PolSci	104	4

- Order dependent; almost like a precomputed result
- Very useful if applicable
- Think if you had to do a count(distinct …)

General Compression

- The benefit depends on the relative I/O and CPU performance
- Tradeoffs keep evolving

Where we are

- We know
  - persistent storage options and characteristics
  - how pages are brought to and replaced from memory
  - how rows are laid-out
  - how row data is stored in pages
  - basics of columnar storage
  - basics of data compression

Operations on a Table

- Insert
- Update, usually requiring finding the rows first
- Delete, usually requiring finding the rows first
- Append
- Bulk insert/update/delete

Operations on a Table

- where c1 = 5 and c2 = 6
- where c1 > 5; or where c1 between 5 and 10
- where c2 like '%cs564%'
- where myudf(c3,c4) = 10
- simply return or aggregate a table
- join a table to another: t1.fk = t2.pk

Operations on a Table

- Equality conditions
- Inequality conditions, including range
- Complex conditions
- Joins, typically FK-PK join: special equality condition
- Scan (full table select, aggregates, etc.)
- Sort (starting point for joins, also aggregate, O-A etc.)

What's Needed

- Ability to insert, delete, update rows
- Find rows meeting a condition
- Read all rows
- Sort rows

Heap Files

- Meet all the requirements
- Great for insert/append
- Great for scan based operations
- But equality conditions, range conditions, and join will require more work

Typical Novel

- No ToC, no index
- Supposed to be read from beginning to end (scan)
- Sometimes published in chapters, so more chapters "appended" to existing ones
- One can find any word/idea by simply reading (scanning) from the beginning

Heap Files Great But ...

- If there are lots of equality, range conditions
- If there is a lot of join of certain kinds (fk-pk)
- Then some organization would be helpful

Typical Textbook

- The main text: Organized by subject matter
  - The Data
- The ToC gives the summary of organization
  - Primary Index, Clustered Index
  - If close in ToC, then close in main text
- Subject index makes it easy to look up particular topics
  - lookup specific key words, closeness in index $\not\Rightarrow$ closeness in text
- Author index makes it easy to look up references to authors in text

What are we looking for?

- Some kind of organization (or order)
- Some kind of a simple ToC idea to find main topics helping with both equality, and range constraints
- Some indices for “equality” constraint lookup on different attributes (“columns”)

File Storage

- Heap/Unordered Files (really, append mostly files)
- Sorted Files - great idea but hard to maintain
  - insert requires either having preplanned gaps or moving things around
  - lookup requires binary search, efficient but not great

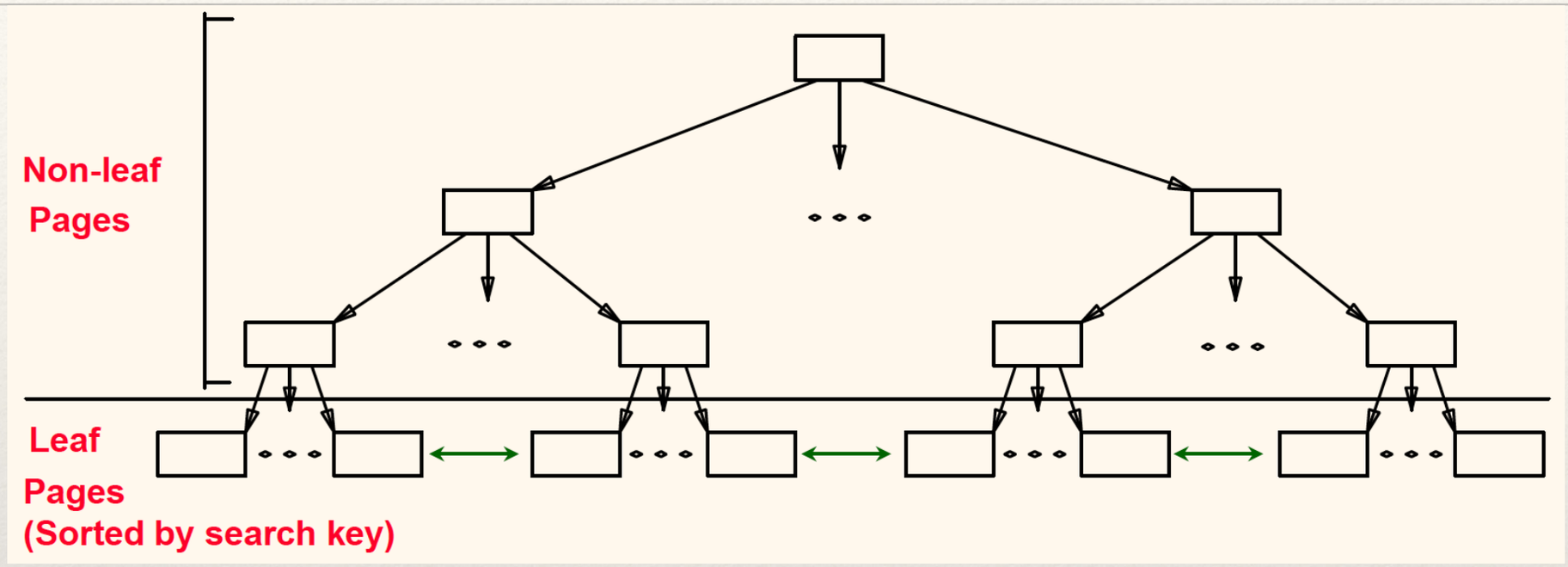
File Storage

- Sorted Files with some sort of ToC idea aka Index

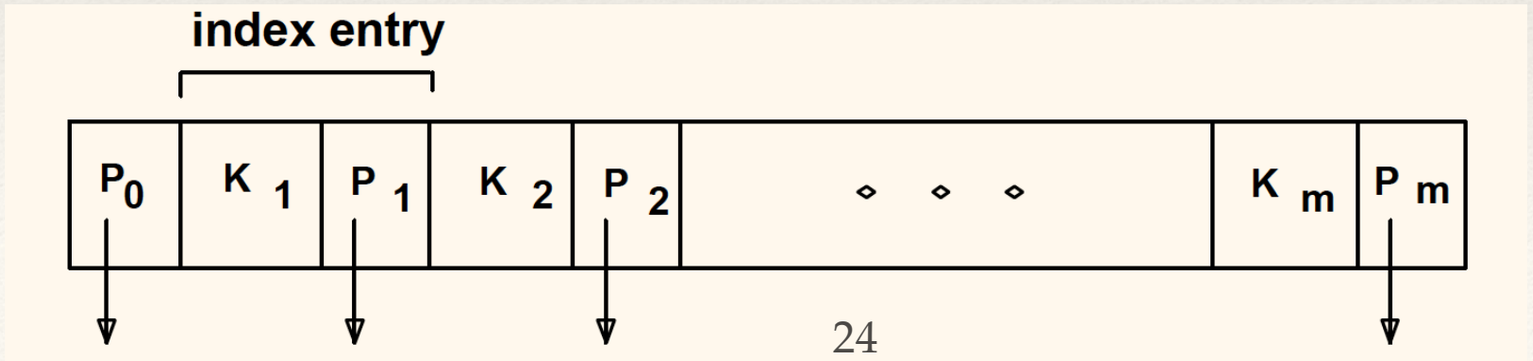
Typical Table

- Main data: organized by some criteria
- Primary/clustered Index: the main index to get to this data
- Secondary indices: pointers to the main data (usually using record-id's (*rid).*
- B+ trees comprise both
  - data in organized (sorted) form
  - the primary index to get to parts of it quickly

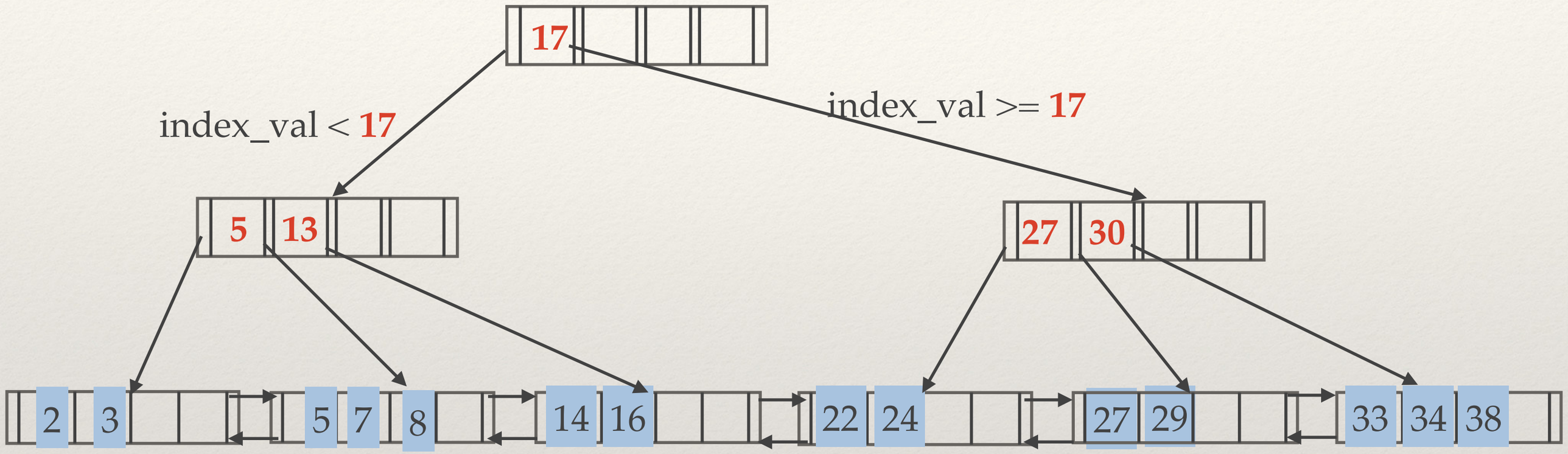
B+ Tree - Preview



- Leaf pages contain data (rows/records) (*data entry*)
- Non-leaf pages have index entries (*search key*)

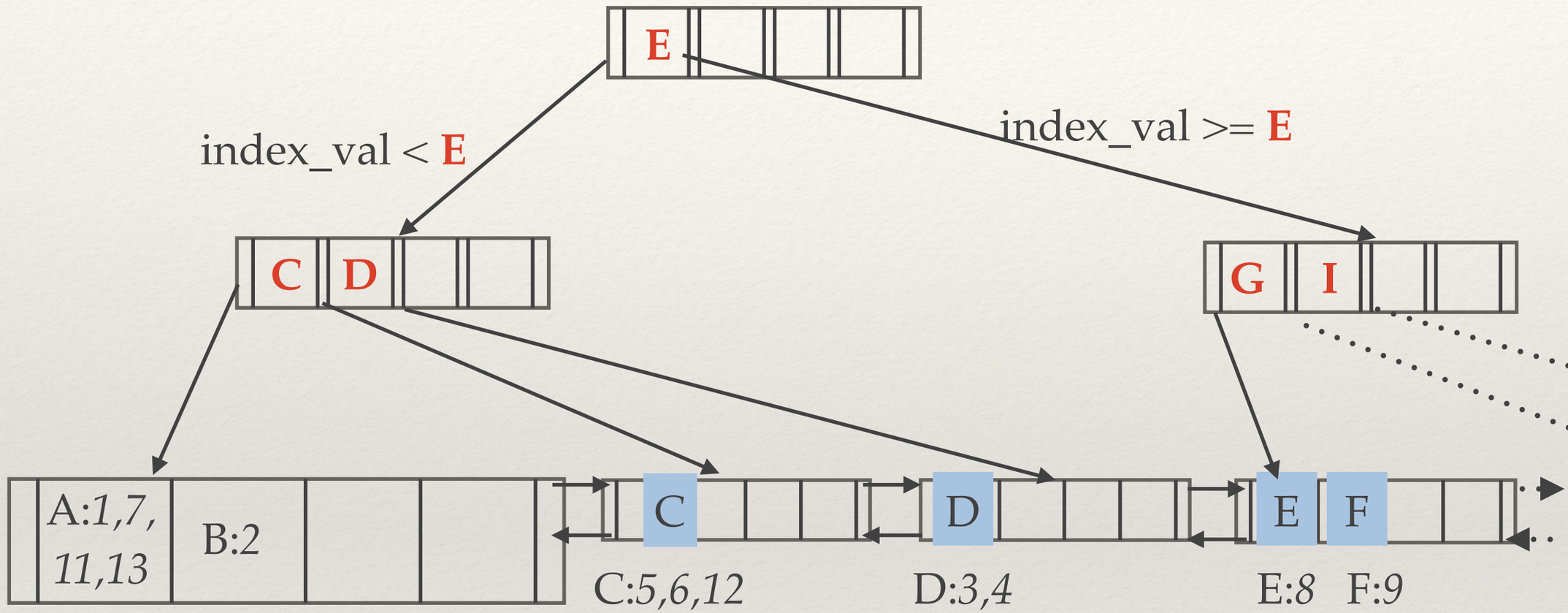


B+ Tree - the ToC or Primary Index



- Find records with index_val = 28?
- Find records with index_val > 15 and < 30?
- insert / delete : find data in leaf, change, may have to propagate changes up the tree

B+ Tree for Secondary Index



rid	C1	C2
1	2	A
2	3	B
3	5	D
4	7	D
5	8	C
6	14	C
7	16	A
8	22	E
9	24	F
10	27	G
11	29	A
12	33	C
13	34	A
14	38	G

- Find records with index_val = B?
  - matching *rid=2*, now go look up *rid=2* in the table
- Find records with index_val between C and D?

B+ Tree is Very Flexible

- Records with index values
  - clustered, primary, like ToC in book, hopefully unique
  - a primary key is usually a good choice
- index value, list of *rid*'s that match record
  - secondary, unclustered, non unique
- index value, *rid* for matching record ⇒ unique
  - secondary, unclustered, unique
  - e.g., can be used to identify other candidate keys in table