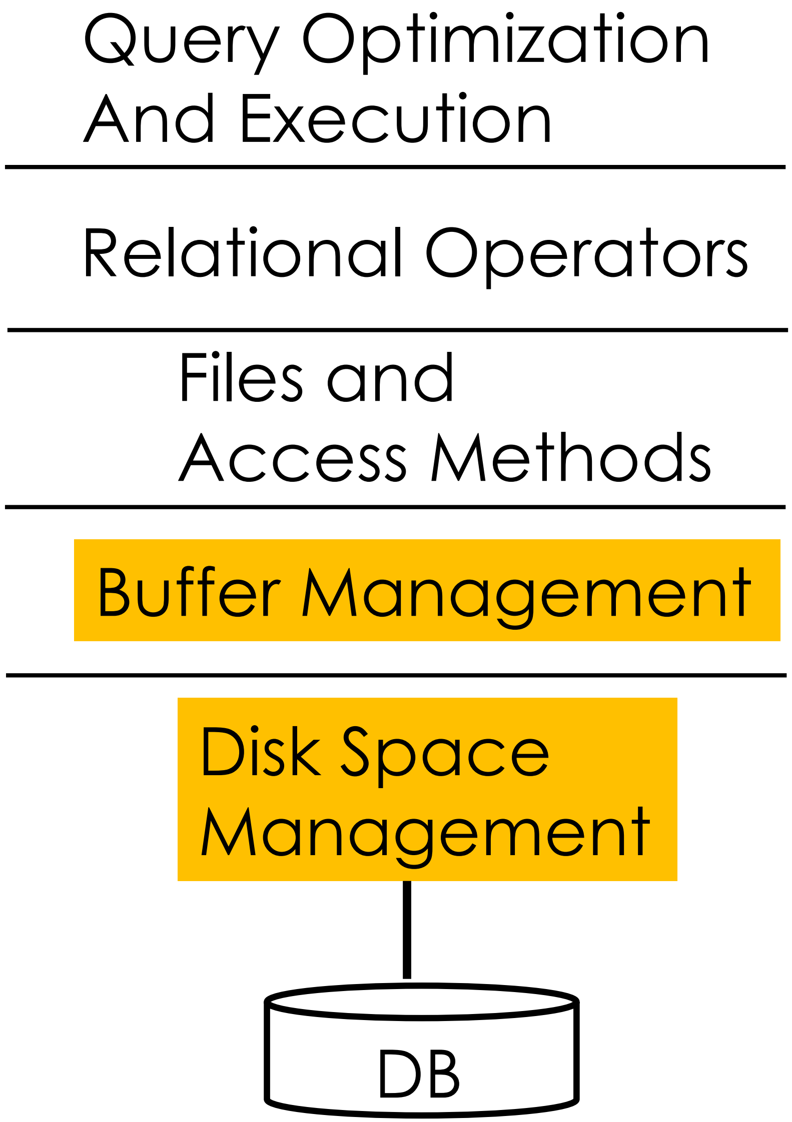
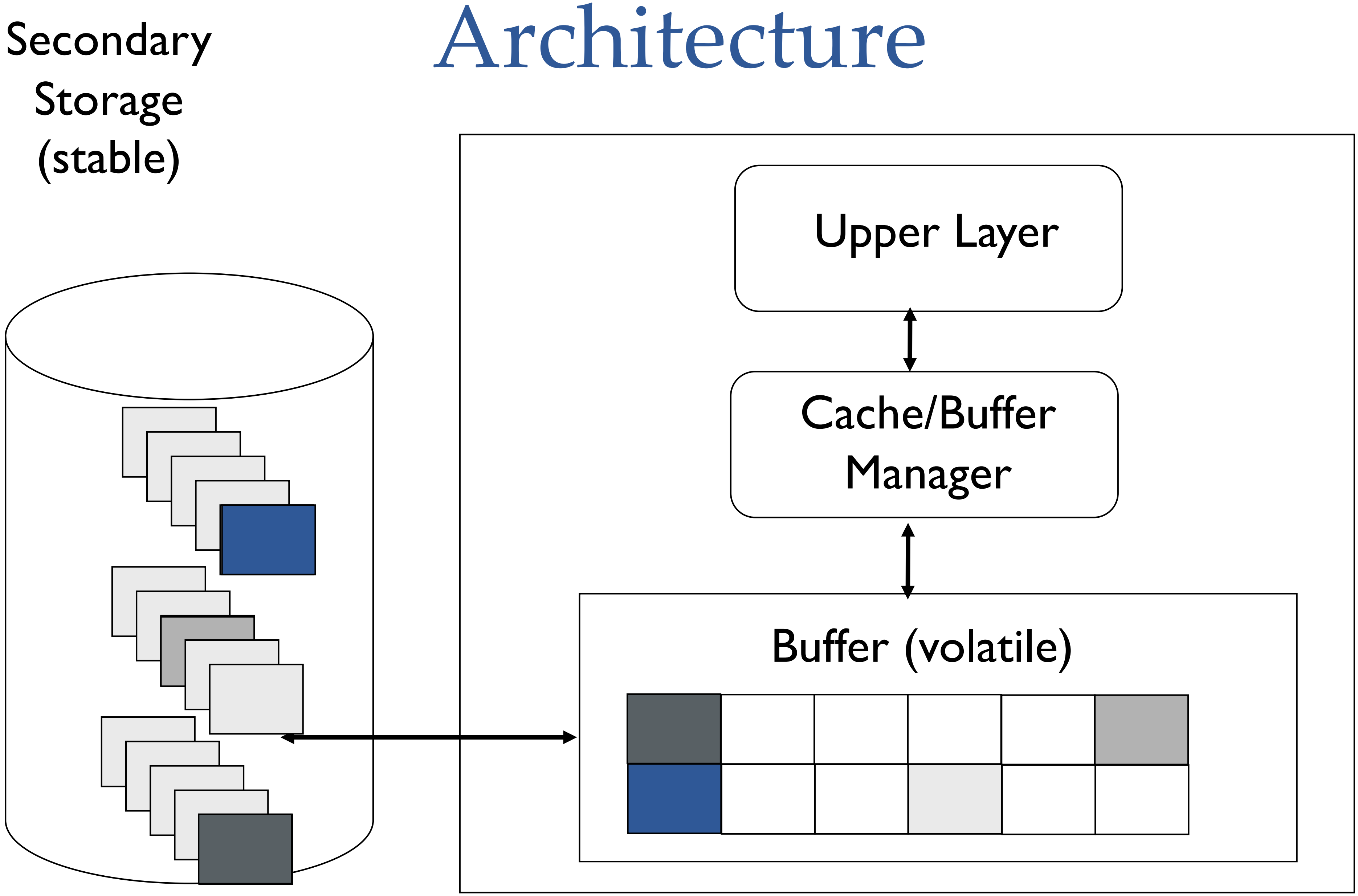
Indexing

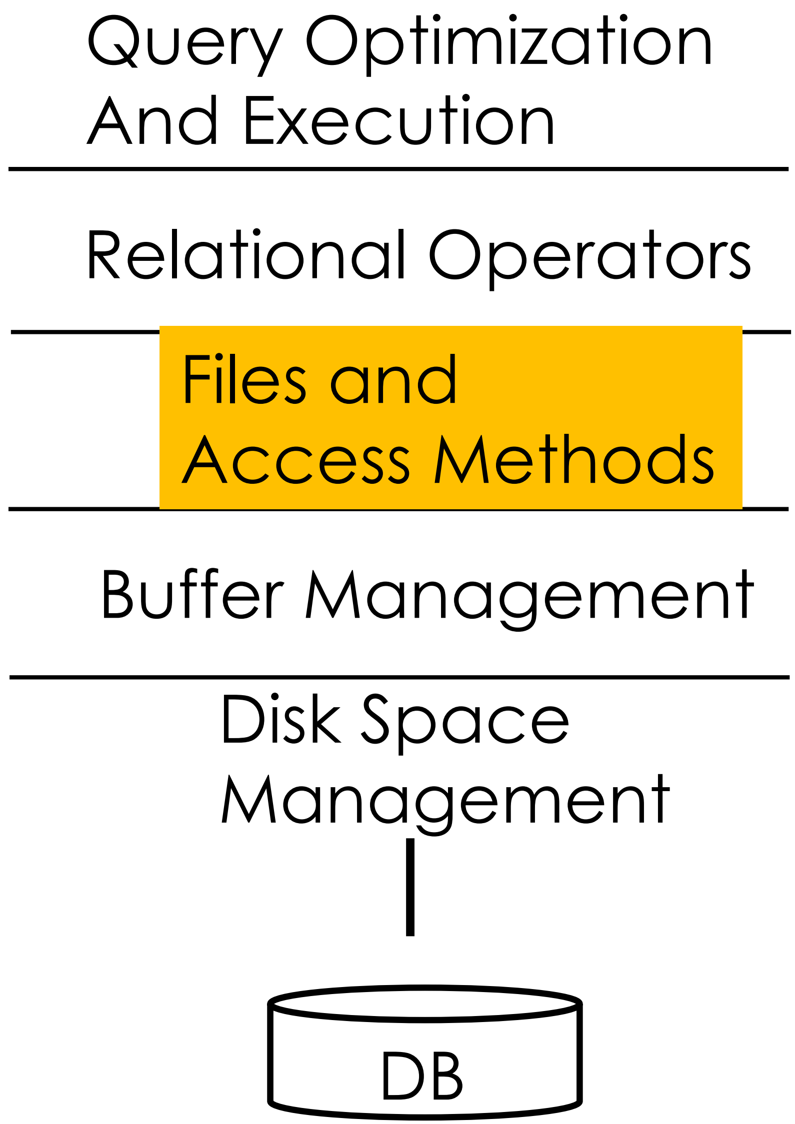
Internals of a DBS I



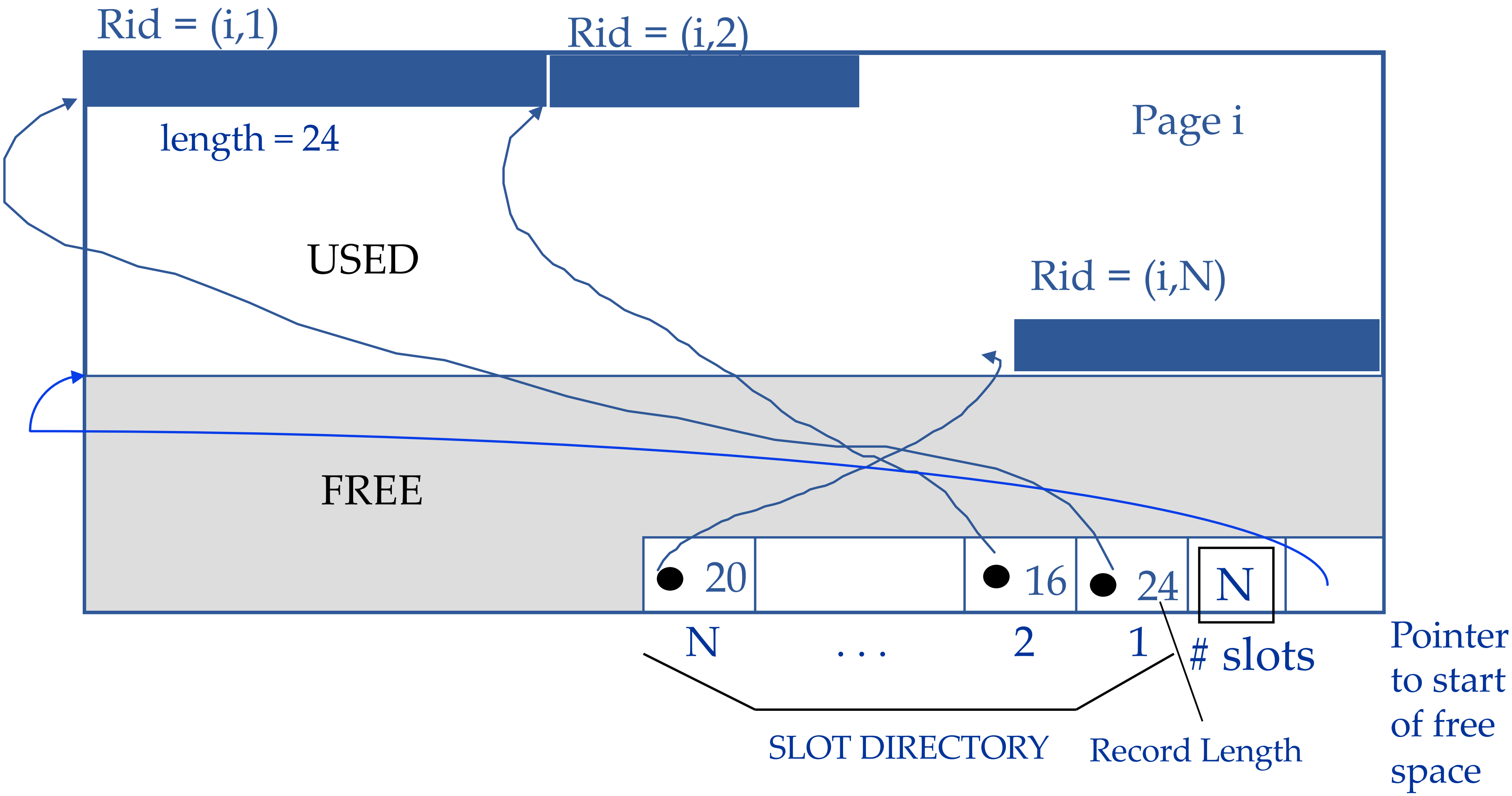
Architecture



Internals of a DBS I



Page Formats: Variable Length Records



- **<u>Record id (rid)</u>** = internal identifier of a record: <page id, slot #>.
- Can move records on page without changing rid;

Typical Operations

- Scan over all records
  - `SELECT * FROM Students`
- Point Query
  - `SELECT * FROM Students WHERE sid = 100`
- Equality Query
  - `SELECT * FROM Students WHERE starty = 2015`
- Range Search
  - ```
    SELECT * FROM Students
         WHERE starty > 2012 and starty <= 2014
    ```
    ```

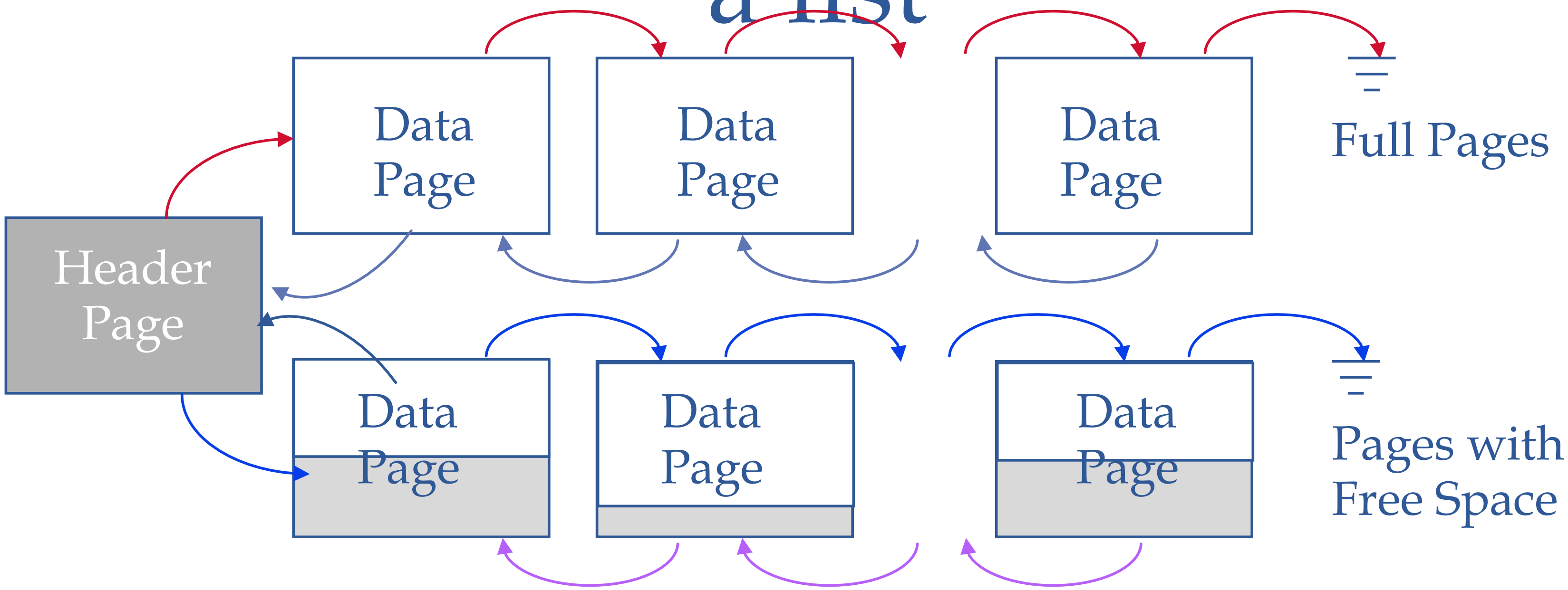
Typical Operations

- Insert
  - `INSERT INTO Students VALUES (23, ‘Bertino”,2016,… )`
- Delete
  - `DELETE FROM Students WHERE sid = 100`
  - `DELETE FROM Students WHERE endyear < 1950`
- Update
  - Delete+insert

Cost Model for Execution

- How should we estimate the costs for executing a statement?
  - Number of I/Os
  - CPU Execution Cost
  - Network Cost in distributed system (ignore for now)
- Assumption in this course
  - I/O cost >>> CPU cost
  - Real systems also consider CPU
- Simplifications
  - only consider disk reads (ignore writes -- assume read-only workload)
  - only consider number of I/Os and not the individual time for each read (ignores page pre-fetch)
  - Average-case analysis; based on several simplistic assumptions.
    - ☛ Good enough to show the overall trends!

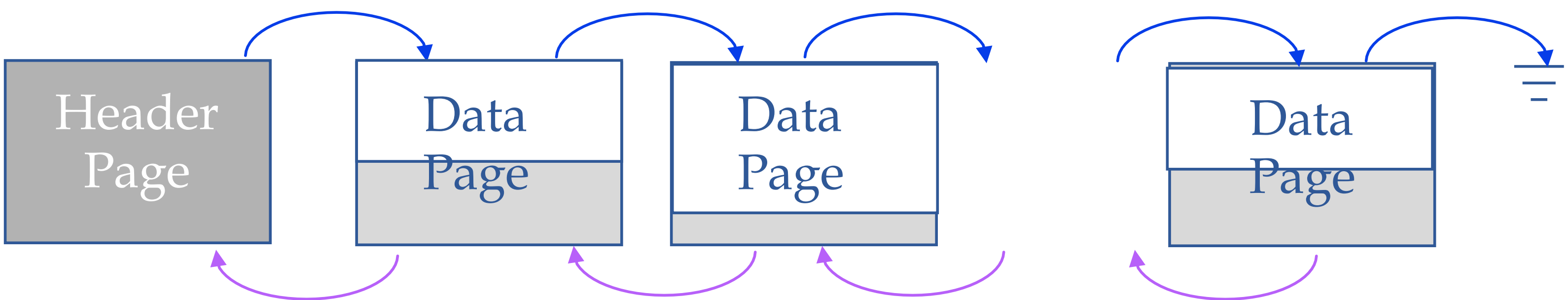
Unsorted heap file implemented as a list

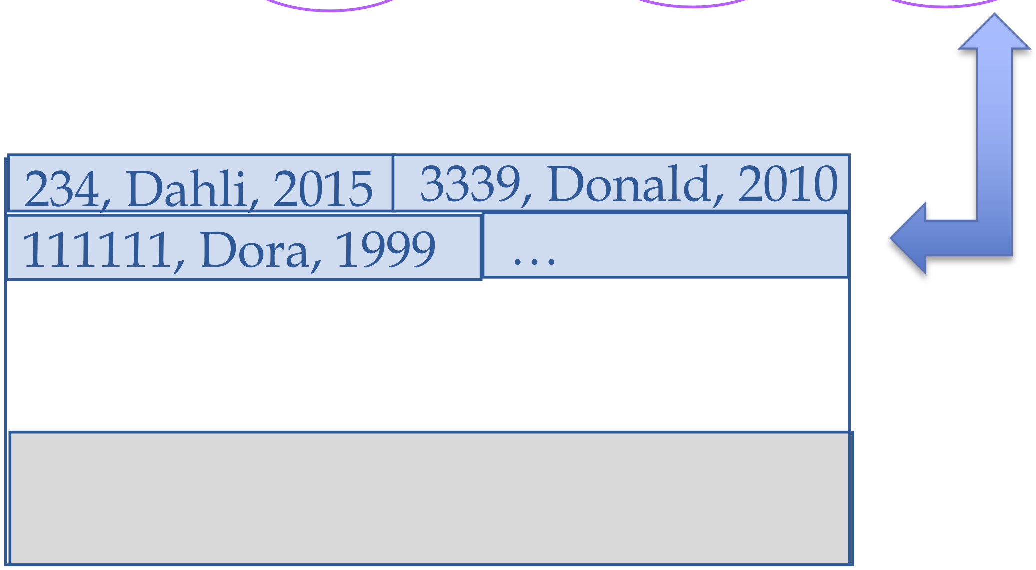


- Good for insert (just find a not full page and append)
- Good for reading entire relation (as storage is compact)

Sorted file

- Records are sorted by one of the attributes (e.g., name).





Zoom

- Reasonably good for equality or range search on sort attribute
- Good for sorted output ordered by sort attribute

Indexes

- Even a sorted file only supports queries on sorted attributes.
- Solution: Build an index for any attribute (collection of attributes) that is frequently used in queries
  - ☆ **Additional information / extra data structure** that helps finding specific tuples faster
  - ☆ We call the collection of attributes over which the index is built the **search key attributes** for the index.
  - ☆ Any subset of the attributes of a relation can be the search key for an index on the relation.
  - ☆ *Search key* is not the same as *primary key / key candidate*

Creating an index in DB2

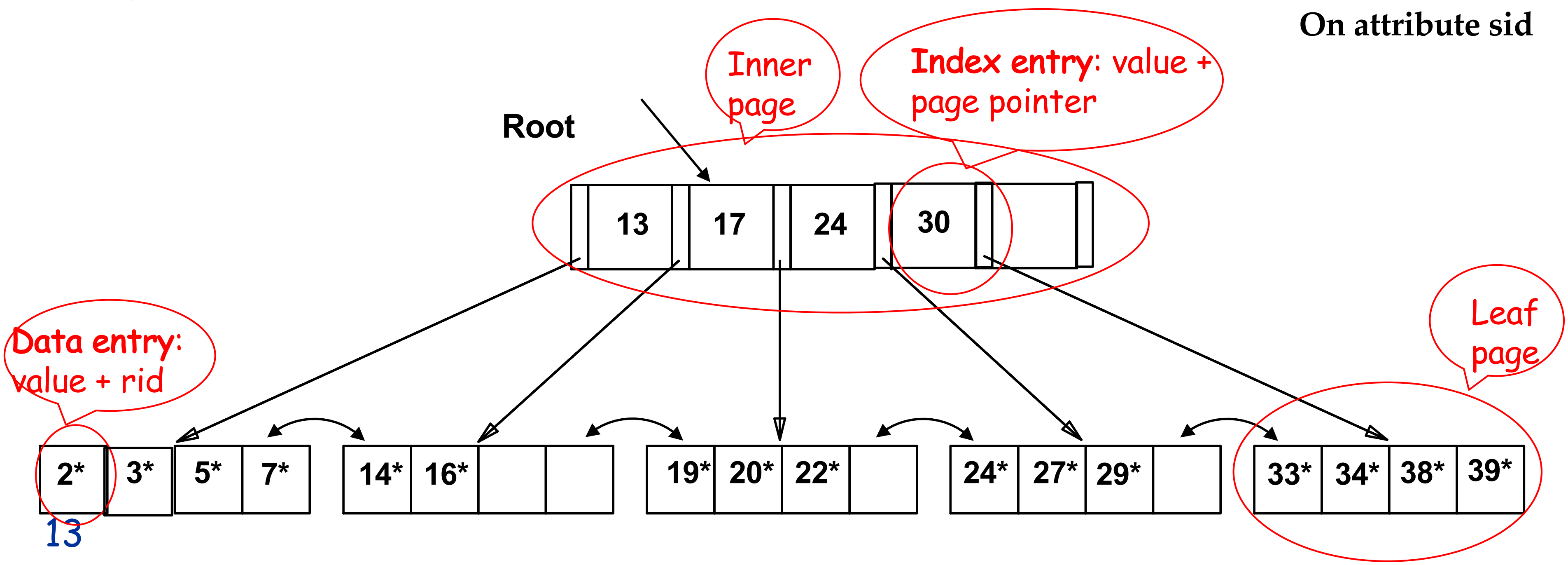
- **Simple**
  - `CREATE INDEX ind1 ON Students(sid);`
  - `DROP INDEX ind1;`

- The search key is sid (it could be any other attribute of the relation)
- An index on sid helps with queries that have a equality condition on sid (and sometimes also helpful for range queries)
  - sid = 58
  - sid <10

B+ Tree: The Most Widely Used Index

- Each node/leaf represents one page
  - ☆ Since the page is the transfer unit to disk
- Leafs contain *data entries* (denoted as k*)
  - ☆ For now, assume each data entry *refers to* one tuple. The data entry consists of two parts
    - Value of the search key (k)
    - Record identifier (rid = (page-id, slot))
  - ☆ That is: data entry is NOT a tuple but a pointer to a tuple
- Root and inner nodes have auxiliary *index entries*
  - A possible value for the search key
  - A pointer to a child

Index for skaters
On attribute sid

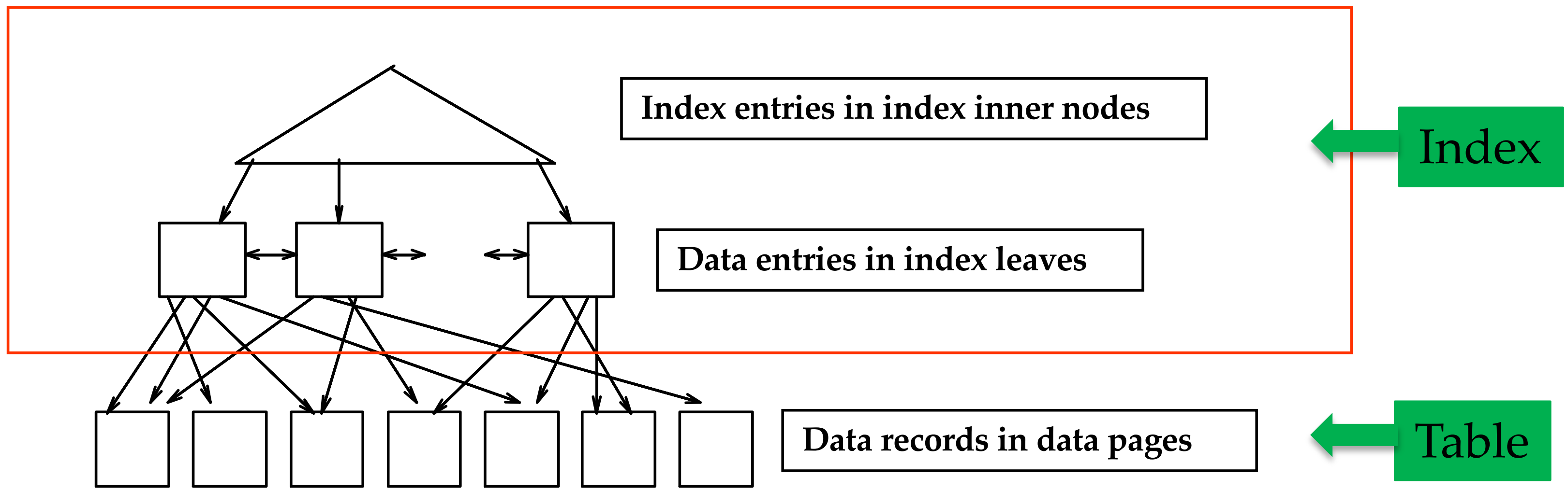


Disclaimer

- Size:
  - Example relation has only 16 tuples
    - You don't build an index for such a relation
  - Think of thousands, and million and more tuples
- Better Example:
  - McGill's student relations
  - Sid = student id = 15-digit number
- Index pages:
  - Inner pages contain hundreds of index entries
  - Leave pages contain hundreds of data entries

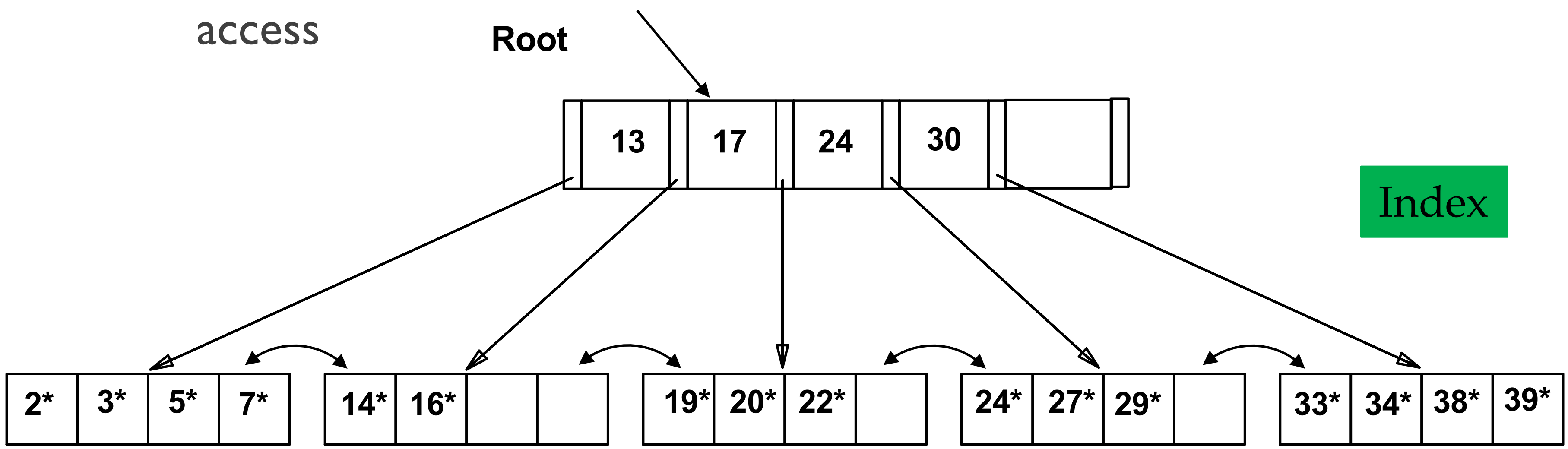
B+ Tree (contd.)

- *height-balanced*.
  - ☆Each path from root to tree has the same height
- F = fanout = number of children for each inner node (~ number of index entries stored in node)
- N = # leaf pages
- Insert/delete at $\log_F N$ cost;
- Minimum 50% occupancy (except for root).



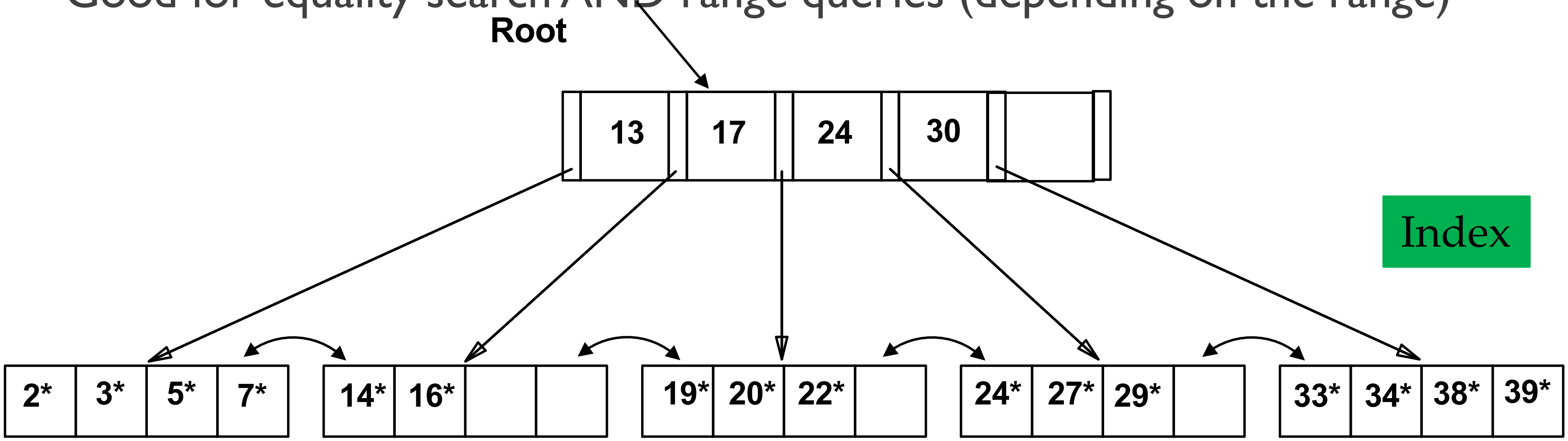
Example B+ Tree

- Example tree has height of 1
- "`Select * from Skaters where sid = 5`"
  - Search begins at root, and key comparisons direct it to a leaf
  - Number of pages accessed:
    - three: root, leaf, data page with the corresponding record
  - Number of I/O:
    - depends of how much of tree is already in the buffer in main memory
    - rough assumption: root and intermediate nodes are always in main memory; index leaves and data pages not in main memory upon first access



Example B+ Tree

- “`Select * from Skaters where sid = 5`”
  - I/O costs:
    - one for leaf page with data entry, one for data page with data record
- “`Select * from Skaters where sid >= 33`”
  - I/O costs:
    - one for leaf page
    - four for data pages with records
- Good for equality search AND range queries (depending on the range)

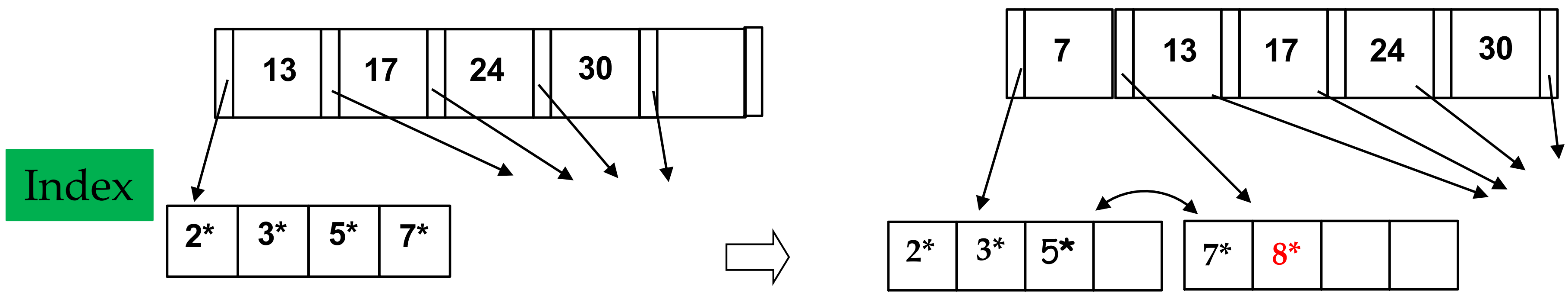


Inserting a Data Entry

- Find correct leaf *L.*
- Put data entry into *L.*
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

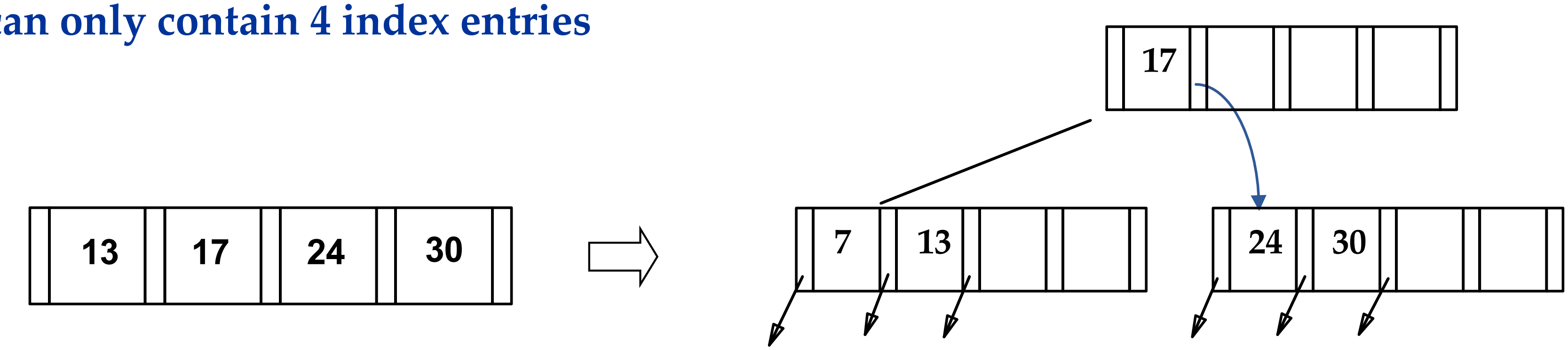
Inserting 8* into Example B+ Tree

Insert into Leaf with leaf split

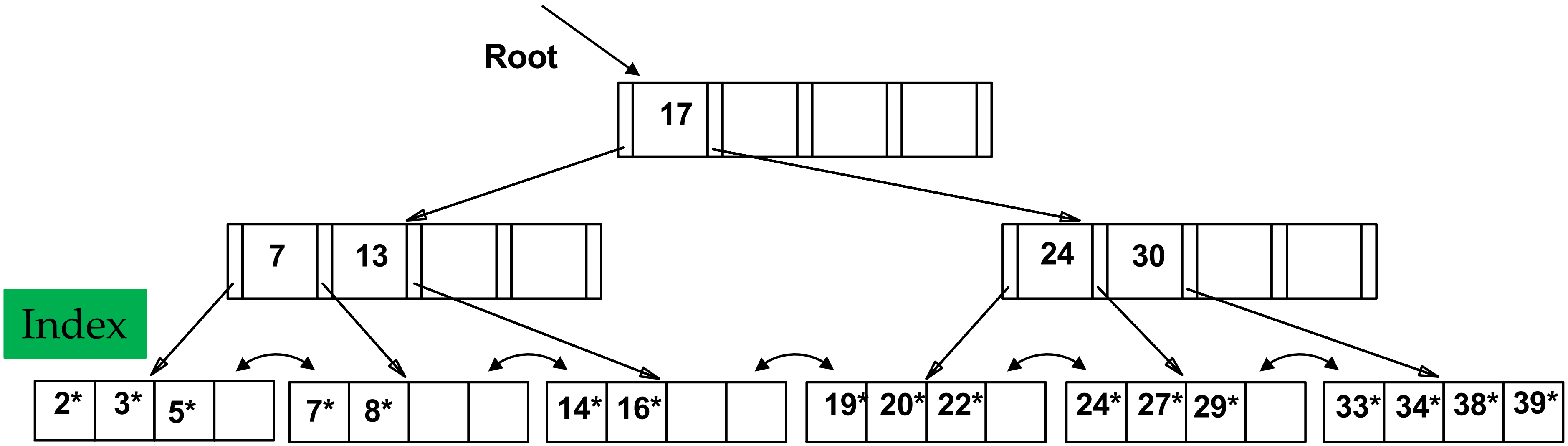


Insert into internal node with node split

Assume that inner pages
can only contain 4 index entries



Example: After Inserting 8*



❖ Notice that root was split, leading to increase in height.

❖ In this example, we can avoid split by redistributing entries; however, this is usually not done in practice.

Data Entry Alternatives: indirect Indexing

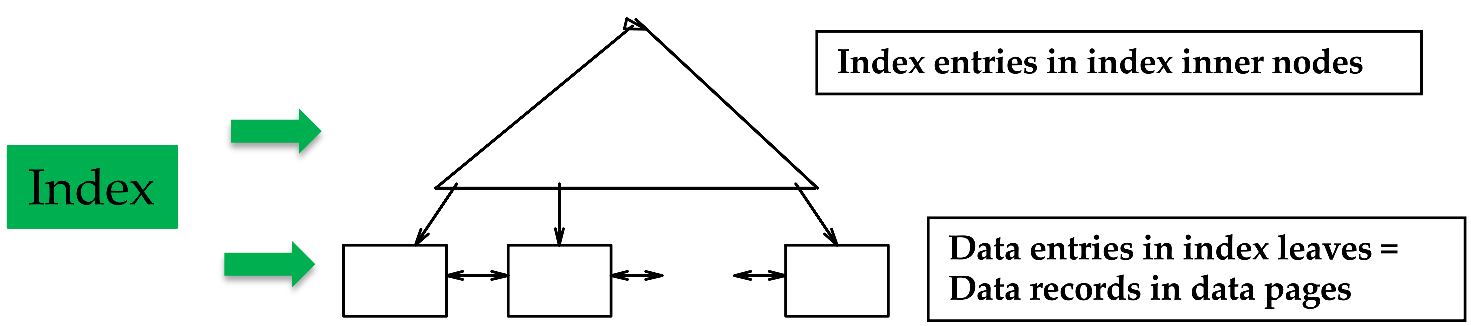
- Indirect Indexing I
  - so far: k* = <**k**, rid of data record with search key value **k**> (indirect indexing)
  - on non-primary key search key: (2022, rid1), (2022, rid2), (2022, rid3), …
    - several entries with the same search key side by side
- Indirect indexing II
  - <**k**, list of rids of data records with search key **k**> (indirect indexing)
  - on non-primary key search key: (2022, (rid1, rid2, rid3,…)), (2023, (rid…
- Comparison:
  - first requires more space (search key repeated)
  - second has variable length data entries
  - second can have large data entries that span a page

Direct Indexing

- Instead of data-entries in index leaves containing rids, they could contain the entire tuple
  - data-entry = tuple
  - no extra data pages
- This is kind of a sorted file with an index on top

Data Entry Alternatives: Direct Index

- Structure
  - <((**k**), full record>
  - Leaf pages contain entire records
    - Data entries = records (not only pointers to them)
    - e.g., index on sid of Skaters
    - (1,lilly,10,16), (2,debby,8,10)…
  - Leafs represent sorted file for data records.
  - Inner nodes above leafs provide faster search
  - At most one direct index per relation
    - Relation can only be sorted by one attribute

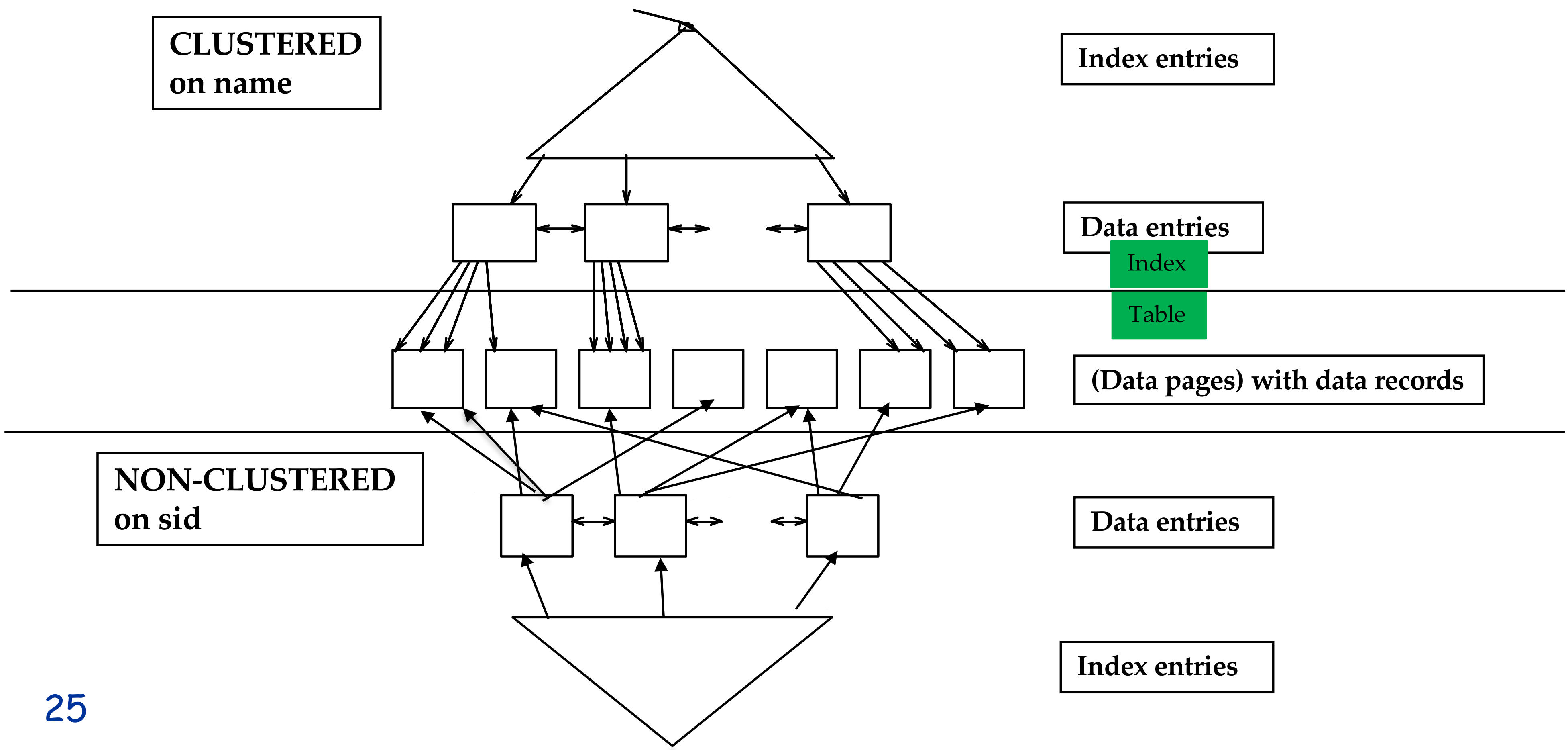


Clustered vs. Non-clustered Index

- Clustered:
  - Relation in file sorted by the search key attributes of the index
- Non-clustered:
  - Relation in heap file or sorted by an attribute different to the search key attribute of the index.
- 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!
- Direct index → clustered index
- Indirect index → can be a clustered index or a non-clustered index

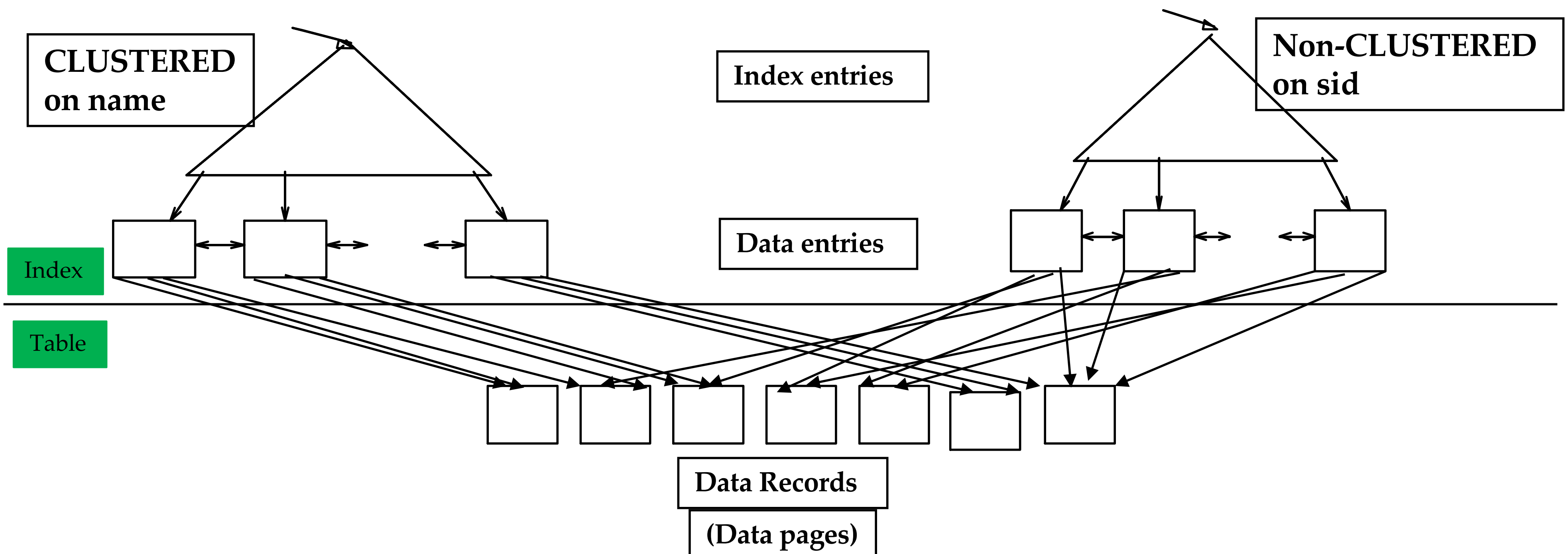
Clustered vs. non-clustered Index

- Example for Students:
  - ☆Indirect clustered index on name
  - ☆Indirect non-clustered on sid



Clustered vs. non-clustered Index

- Same Example – different visualization
  - Indirect clustered index on name
  - Indirect non-clustered on sid



Primary/Unique Indices

- Primary vs. secondary: If search key contains primary key, then called primary index.
  - ☆ *Unique* index: Search key is primary key or unique attribute.
- ☆ Secondary index: search key is not unique

B+-tree cost example: Given

- About the relation
  - Relation R(A,B,C,D,E,F)
  - A and B are int (each 4 Bytes), C-F is char[40] (160 Bytes)
  - Values of B are within [0;19999] uniform distribution
  - 200,000 tuples
- About the unsorted heap file
  - Assume each data page has 4 K (4000 Bytes) and is around 75% full
    - I know, I know: with fixed sized records we don't need to leave space, but in real life records don't have fixed length and we should leave space….(and 75% makes it 3K – so a nice number)
  - The size of an rid = 10 Bytes
- About the index to be built (on B)
  - An indirect index alternative II (one data entry per different value of the search key)
  - An index page has 4K
  - Index pages are filled between 50% - 100% (this always holds!!)
    - Let's assume that they are on average 75% full
  - The size of a pointer in intermediate pages (i.e., a page identifier): 8 Bytes

Size of B+tree: To calculate

- **Number of data pages in the heap file**
  - size of the tuple = 168
  - Number of tuples on a page
    - 4000 * 0.75 / 168 ≈ 18 tuples in a page on an average.
  - Number of pages 200000 / 18 ≈ 11111 pages in total.
  - OR: 200000 * 168 / 3000 = 11200
- **The number of leaf pages**
  - Number of data entries:
    - distinct values of B = 20000
  - Number of records with same value / number of rids in a data entry
    - 200000 / 20000 = 10 records per val of B
  - Size of data entry
    - sizeof(B) + 10 rids * sizeof(rids)
    - 4 + 10 * 10 = 104 bytes/data entry
  - How many leaf pages?
    - How many data entries per page
      - 4000 * 0.75 / 104 ≈ 28.846 data entries/page on an average.
      - Or min of 4000* 0.50 / 104 and max of 4000* 1.0 / 104
    - How many leave pages: 20000/29 ≈ 690 leaf pages.
  - How many leaf pages alternative calculation
    - Number of data entries * size of data entry / occupied space on leaf page
    - 20000 * 104 / 3000 = 693

Size of B+tree: To calculate

- **The height of the tree**
  - Size of an index entry in intermediate node
    - 4 + 8 = 12 bytes
  - Max. number of index entry for intermediate page:
    - 4000 / 12 = 333
    - A single root node cannot hold pointers to all leaf pages. Thus, height > 1
  - intermediate nodes at least 50% (167 index entries) max 100% (333 index entries) often 75% (250 index entries)
  - They have a total of around 690 pointers to leaf nodes
  - Thus, likely 3 intermediate nodes right above the leaf pages (to manage 690 leaves).
  - One node above then to hold pointers to the 3 intermediate pages - root.
  - height = 2 (the number of edges from root to a leaf node)

B+ Trees in Practice

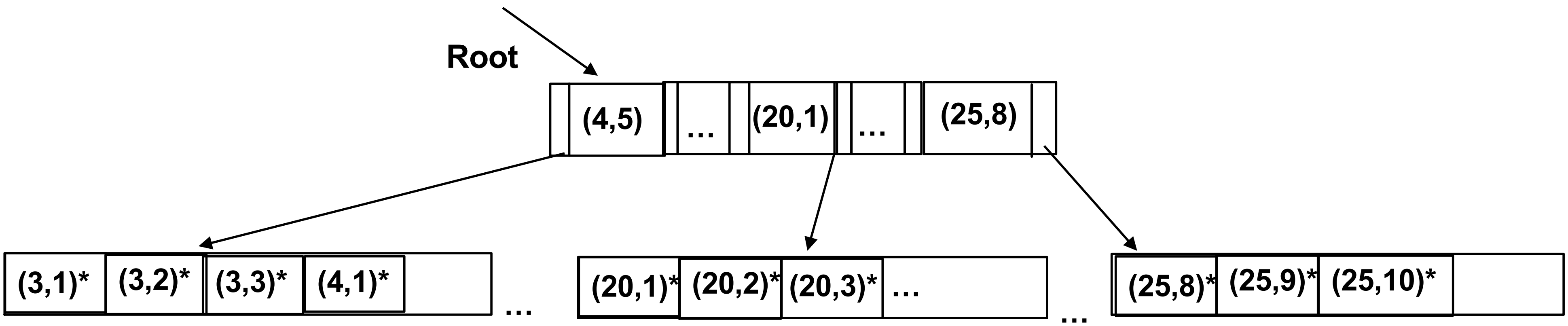
- Typical max. number of index entries in inner nodes: 200
  - (I.e., an inner node has between 100 and 200 index entries)
  - Typical fill-factor: 67%.
  - average fanout = 133
- Leaf nodes have often less entries since data entries larger (rids)
  - For simplicity: assume 100
- Typical capacities (roughly)
  - Height 1: 133 * 100 = 13,300 records (just the root and leaves)
  - Height 2: 133^2 * 100 = 1,768,900 records
  - Height 3: 133^3 * 100 = 235,263,700 records
- Can often hold top levels in buffer pool:
  - Level 1 (root) = 1 page = 4 Kbytes
  - Level 2 = 133 pages = 0.5 Mbyte
  - Level 3 = 17,689 pages = 70 MBytes

Multi-attribute index

- Index on `Skaters(age,rating);`
- Order is important:
  - Here data entries are first ordered by age
  - Skaters with the same age are then ordered by rating
  - assume youngest skater is 3, oldest is 25 (list of rids: *):
    - the leaf pages then would look like:

(3,1)*	(3,2)*	(3,3)*	(4,1)*		…	(20,1)*	(20,2)*	(20,3)*	…	…	(25,8)*	(25,9)*	(25,10)*	

What does it support



- What does it support?
  - `SELECT * FROM Skaters WHERE age = 20 AND rating = 5;`
    - Yes
  - `SELECT * FROM Skaters WHERE age = 20 AND rating < 5;`
    - Yes
  - `SELECT * FROM Skaters WHERE age = 20;`
    - yes
  - `SELECT * FROM Skaters WHERE rating < 5;`
    - No
  - `SELECT * FROM Skaters WHERE age = 20 OR rating <=5;`
    - No

Index in DB2

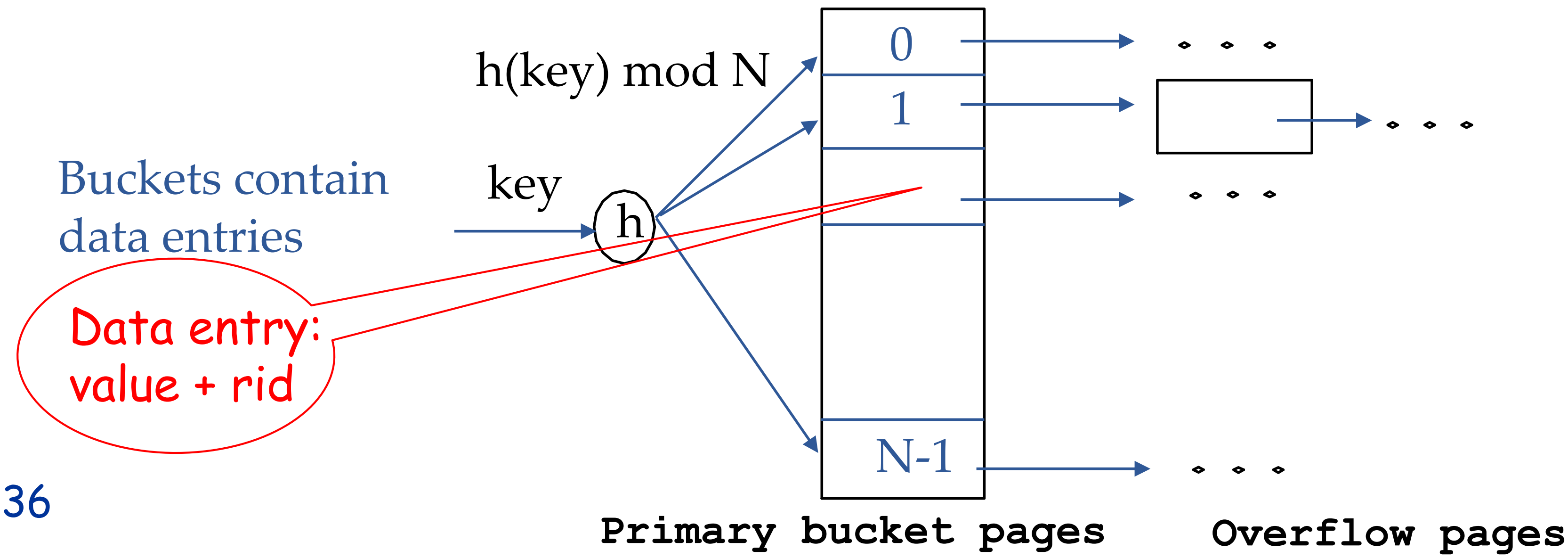
- Simple
  - `CREATE INDEX ind1 ON Students(startyear);`
  - `DROP INDEX ind1;`
- Clustered index
  - `CREATE INDEX ind2 on Students(name) CLUSTER`
- Index also good for referential integrity (uniqueness)
  - `CREATE UNIQUE INDEX indemail ON Students(email)`
- Multi-attribute index
  - `CREATE INDEX ind3 on Students(name,startyear)`
- Additional attributes
  - `CREATE UNIQUE INDEX ind1 ON Students(sid) INCLUDE(name)`
  - Index only on sid
  - Index entries in inner pages only contain search key attribute sid
    - only conditions on index attribute are supported
  - *Data entries* in the leaf pages *additionally* contain name;
    - key value (sid) + name + rid
  - Why?
    - `SELECT name FROM Students WHERE sid = 100`
    - Can be answered without accessing the real data pages of Students relation!

Summary for B+-trees

- Tree-structured indexes are good for equality searches and often work well for range-searches,
- High fanout (**F**) means depth rarely more than 3 or 4.
- Can have several indices on same tables (over different attributes)
- Most widely used index in database management systems because. One of the most optimized components of a DBMS.

Static Hashing

- Similar to standard hashing but with pages as the unit of storage (compare with array-based main memory implementation)
- Decide on a number N of buckets at index creation
- Allocate one primary page per bucket
- Overflow pages for individual buckets are created later as needed
- Buckets contain data entries (same as leaf pages in tree)
- let k be the search key of the index, h a hash function
  - **h**(*k*) mod N = bucket to which data entry with key *k* belongs.



contd

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance.
  - Several optimizations developed such to handle scale dynamically (e.g., extensible and linear hashing)