Principles of Query Execution

Zachary G. Ives
University of Pennsylvania

April 16, 2003

Today

- Query execution architectures
- Data manipulation toolkit
  - “Heaps” and iteration
  - Sorting
  - Hashing
- Tables, clustering, and indices

Reminder: project proposals due Wednesday!

The Execution Engine

- The “cardiopulmonary system” of answering queries
  - Causes data to flow from sources to output
  - The optimizer is the “brain”!
- Input: physical query plan, set of sources
  - (How is physical plan different from logical?)
  - Plan is typically a tree (but can be a graph)
- Engine and phys. plan schedule execution of operators
  - What goes in sequence, what goes in parallel?
  - What operations can be distributed or parallelized?
  - “Push” vs. “pull”

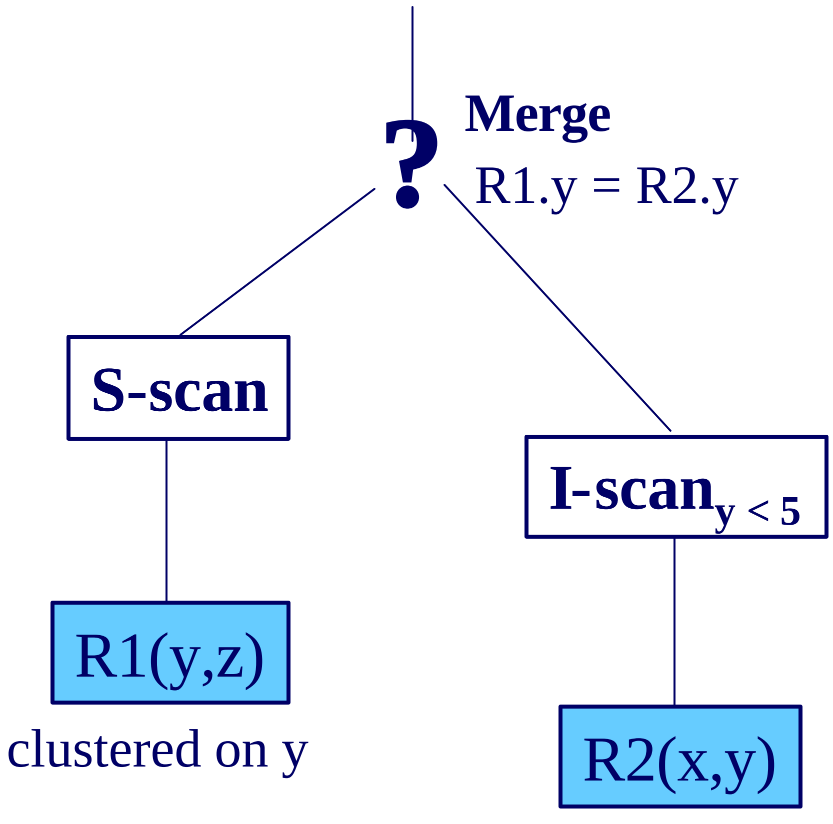
Some Scheduling Possibilities

- Series vs. parallel:
  - Operators execute sequentially (blocking)
  - Operators are pipelined (note effect on state)

- Scheduling:
  - Operators are input-driven (push)
  - Operators get separate threads
  - Operators are demand-driven (iterator)

The Iterator Model in Action

- Methods: open, next, close()



Processing Data Naively

- Relation may be arbitrarily laid out on disk
  - Often called a “heap file” – but not as in the heap data structure
- How can we process it?
  - Iterate through every tuple and test
  - Iterate through every page of tuples and test (when is this different?)

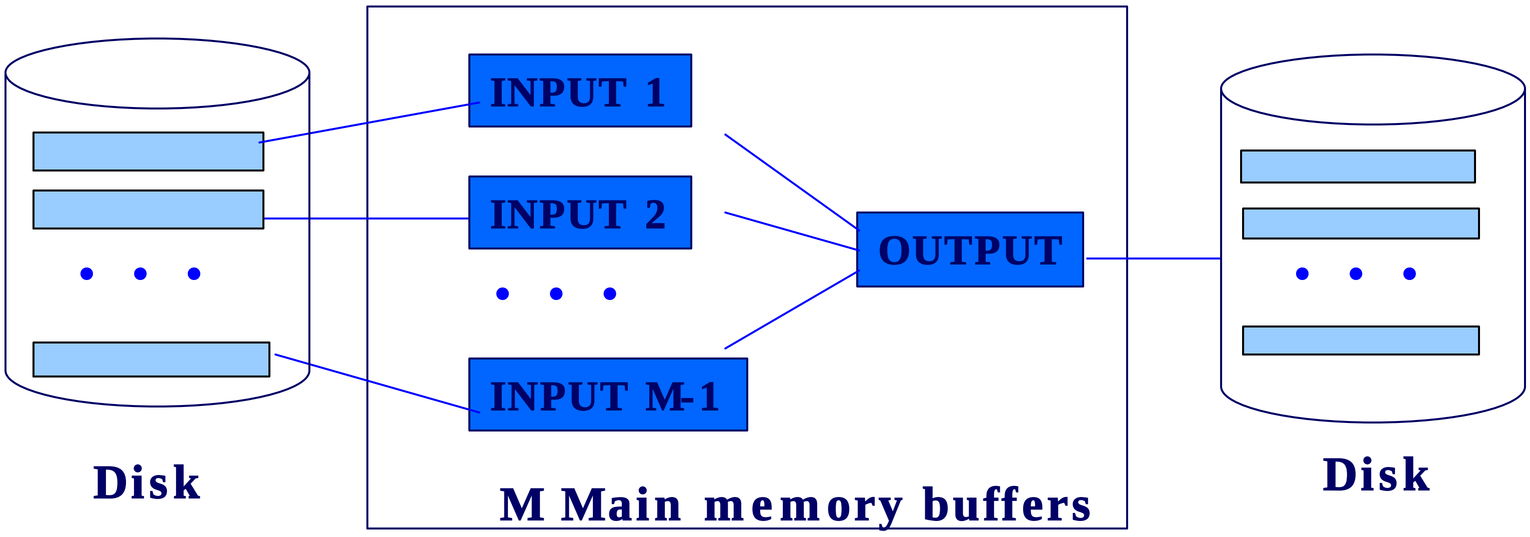
  - Or be smarter in laying out the data…

Sorting

- How do we do sort an R-page table, given *M* pages' worth of memory?
  - We all learned quicksort for in-memory sorts
    - Pick a split point, partition data above and below it
  - Can also do replacement selection
    - Heap-based – sort of like an incremental heapsort
    - Average run file is about 2*M*;
      $\lceil R/2M \rceil + 1$ expected runs
  - But that's only for one run… How do we combine runs efficiently?

General External Merge Sort

- To sort a file with *R* pages using *M* buffer pages:
  - Pass 0: use *M* buffer pages. Produce $\lceil R / M \rceil$ sorted runs of *M* pages each.
  - Pass 2, …, etc.: merge *M-1* runs.



Cost of External Merge Sort

- Number of passes: $1+\left\lceil \log_{M-1}\left\lceil R/M \right\rceil \right\rceil$
- Cost = 2R * (# of passes)
- With 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108/5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22/4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Hashing

- Sorting divides data using physical properties, then combines using logical properties
- Hashing divides data using logical properties (hash key) and then chains them in buckets
- How do we hash an R-page table, given *M* pages' worth of memory?
  - Avoidance: pre-partition the data into *R* / *M* smaller units, then execute each
  - Resolution: partition after we run against bounds

Hybrid Hashing

- R-page table, given *M* pages' worth of memory in F hash buckets:
  - Assign K partitions, each expected to be of size M
  - Leaves M – (K+1)C buffers for hashing
- May need to hash recursively, and skew may affect this
  - Sometimes revert to other algorithms when skew is a problem

Reading from Disk

- Simple sequential scan
- Associative access: indices
  - An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.
  - Hash index – what do you think this looks like?
  - B+ tree
  - Bitmap index

Alternatives for Data Entry k* in Index

- Three alternatives:
  - Data record with key value **k**
    - ✓Clustered -> fast lookup
    - Index is large; only 1 can exist
  - <**k**, rid of data record with search key value **k**>, OR
  - <**k**, list of rids of data records with search key **k**>
    - ✓Can have secondary indices
    - ✓Smaller index may mean faster lookup
    - Often not clustered -> more expensive to use
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.

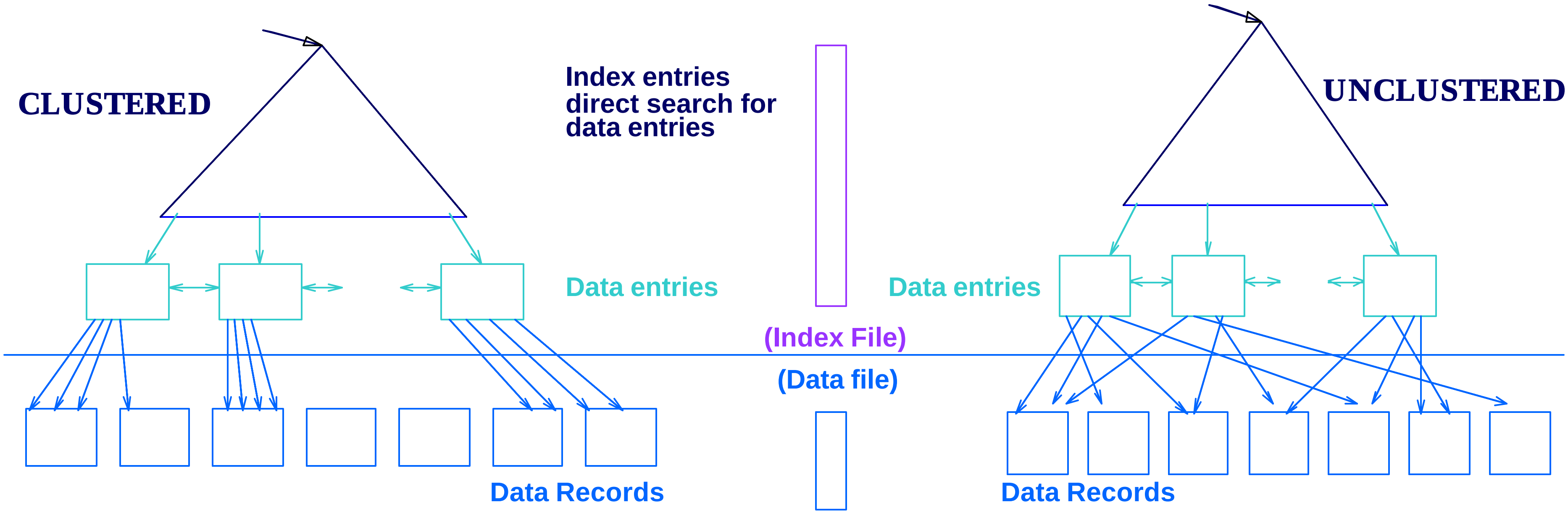
Classes of Indices

- *Primary* vs. *secondary*: primary has primary key
- *Clustered* vs. *unclustered*: order of records and index approximately same
  - Alternative 1 implies clustered, but not vice-versa.
  - A file can be clustered on at most one search key.
- *Dense* vs. *Sparse*: dense has index entry per data value; sparse may “skip” some
  - Alternative 1 always leads to dense index.
  - Every sparse index is clustered!
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.

Clustered vs. Unclustered Index

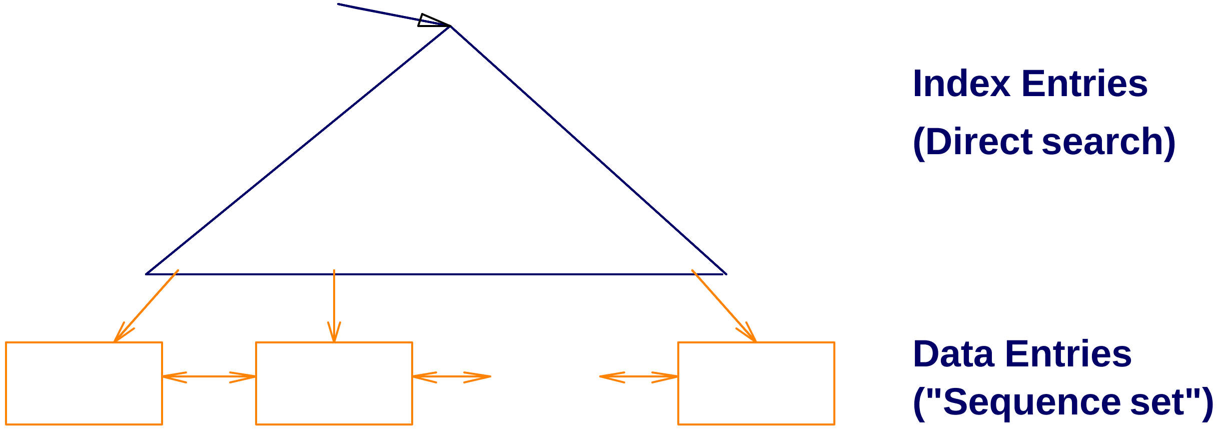
Suppose Index Alternative (2) used, records are stored in Heap file

- Perhaps initially sort data file, leave some gaps
- Inserts may require overflow pages



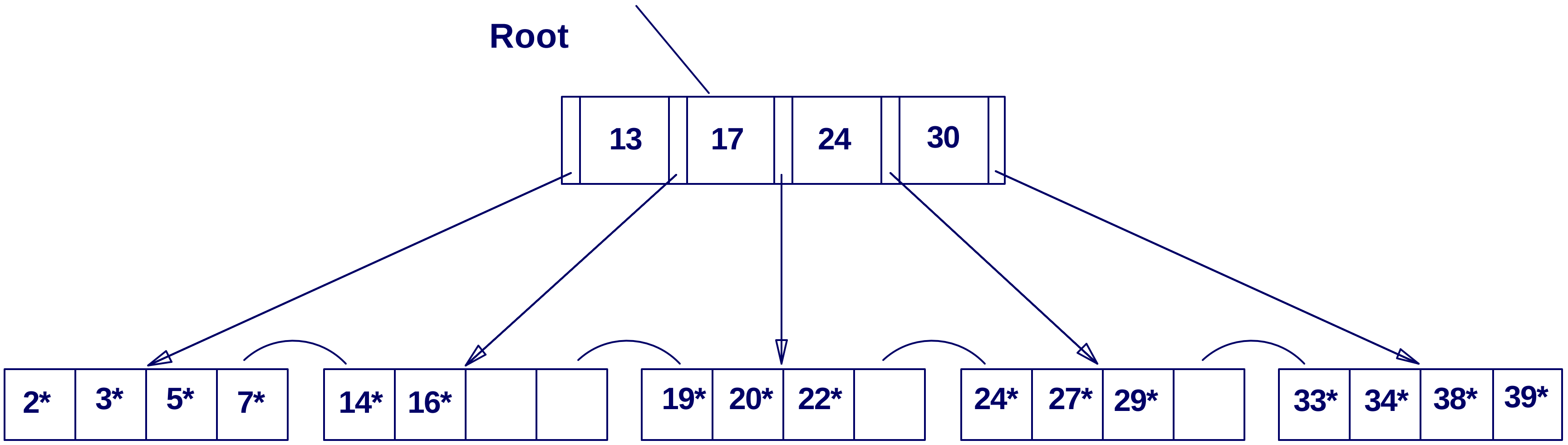
B+ Tree: Our Favorite Index

- Insert/delete at log $_F$ N cost
  - (F = fanout, N = # leaf pages)
  - Keep tree *height-balanced*
- Minimum 50% occupancy (except for root).
- Each node contains **d** <= *m* <= 2**d** entries.
  d is called the *order* of the tree.
- Supports *equality* and *range* searches efficiently.



Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf.
- Search for 5*, 15*, all data entries >= 24* ...



- *Based on the search for 15*, we <u>know</u> it is not in the tree!*

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 1334 = 312,900,700 records
  - Height 3: 1333 =   2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 =   1 page =   8 Kbytes
  - Level 2 =   133 pages =   1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Bitmap Indices

- Primarily useful for discrete values, indices on multiple attributes
  - A bit for each possible value of an attribute
  - Example:
    sex ∈ {M, F}; status ∈ {ugrad, grad, fac}
    PennCIS(ID, name, sex, status)

sex_bmp

MF
10
01
10

status_bmp

UGF
010
010
001

PennCIS

ID	name	sex	status
1	Peng	M	G
50	Kit	F	G
99	Zack	M	F

Wrapping Up

- Today we saw a “toolkit” of techniques for query execution
  - Sorting and hashing to speed up processing of data
  - Need strategies for larger-than-memory operation
- Associative access methods for retrieving data
  - Hash indices
  - B+ trees
  - Bitmap indices
- Wednesday: putting these to use in real algorithms!