Index Files

Alvin Cheung

Aditya Parameswaran

R & G - Chapter 9-10

Connecting Back to the Storage Layer

- So far, we have been talking about a B+tree index pointing to unordered pages in a heap file
- This is not the only approach we can take.

- We'll talk about various alternatives for the:
    - Leaf nodes (the interface between index and the data)
    - Heap file (the actual data)

Three basic alternatives for leaf nodes

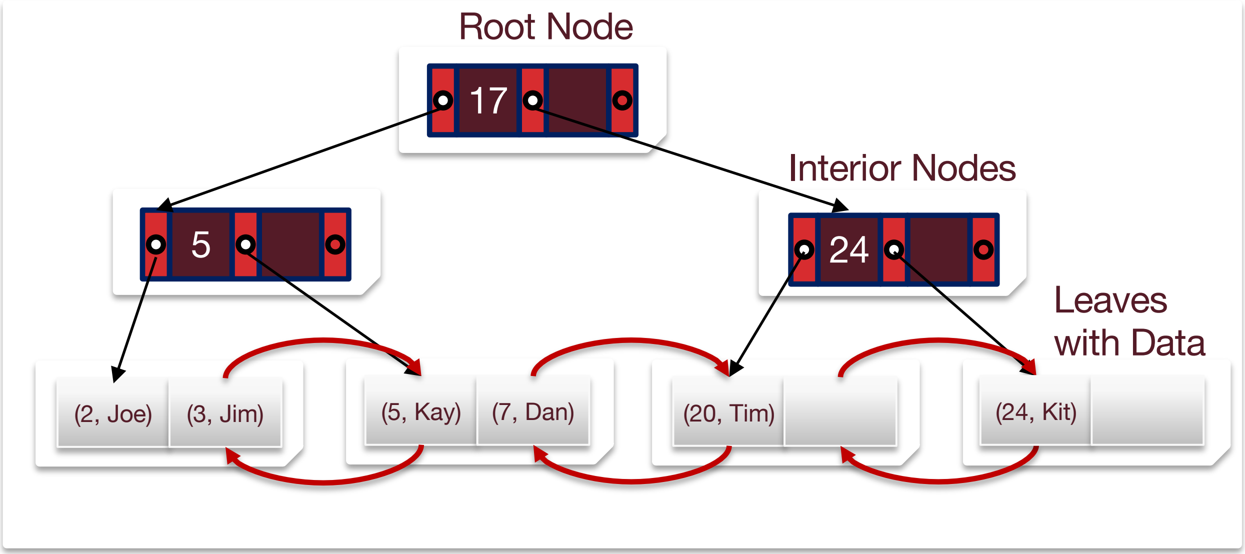
- Also applies for data entries for other types of indexes
- We’ll look in the context of B+-trees, but applies to any index

- Three basic alternatives (Textbook uses same numbering!)
  - Alternative 1: By Value
  - Alternative 2: By Reference ⬅ *this is what we’ve already seen*
  - Alternative 3: By List of references

Alternative 1: By Value

- Leaf pages store records directly
  - No need to follow pointers

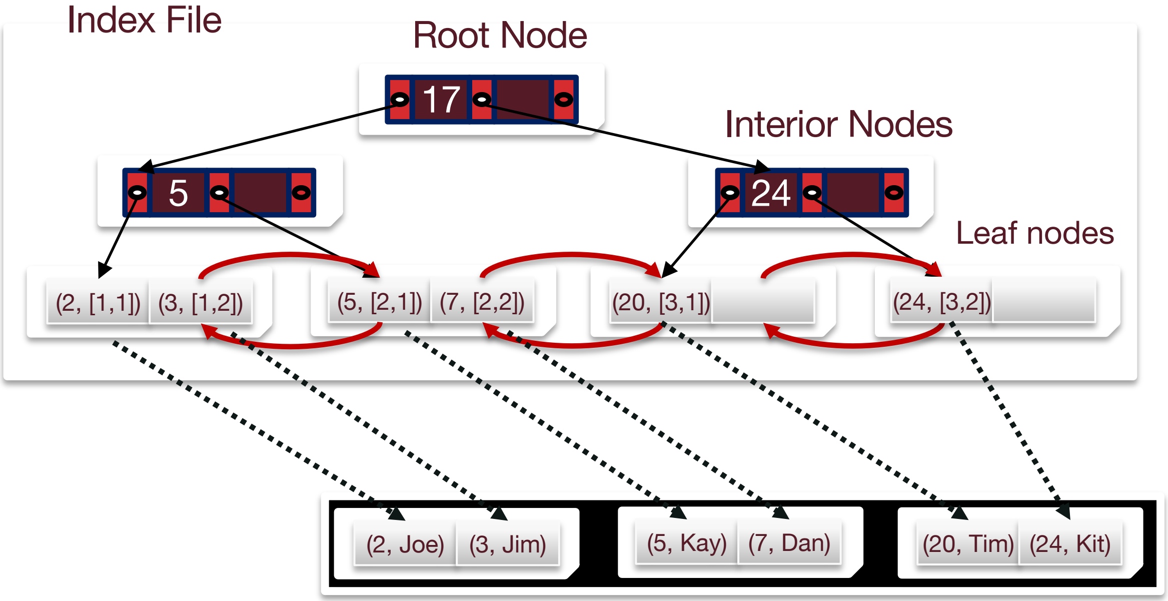
uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit



Alternative 2: By Reference Pairs

- For each **k**, store recordId of matching data record as pairs
  - Each entry in leaf: <k, recordId>
    - Recordid = [page id, slot id]
  - We used this previously

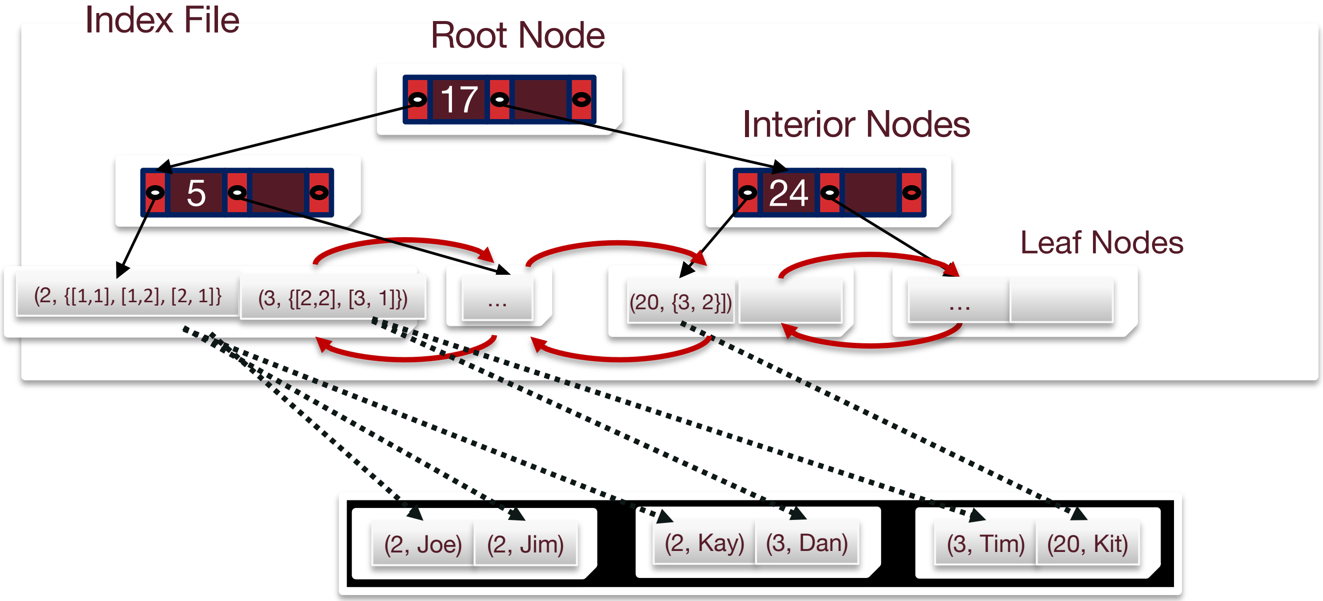
uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit



Index Contains
(Key, Record Id)
Pairs

Alternative 3: By Reference List

- For each k, store recordIds of matching records as a list
  - Each leaf entry: <**k**, {list of rids of matching data records}>
  - Alternative 3 more compact than alternative 2
    - Very large rid lists can span multiple blocks, needs bookkeeping to manage that



Index Contains
(Key, {list of record Id}) Pairs

By Value vs. By Reference

- Both Alternative 2 and Alternative 3 index data *by reference*

- If we want to support multiple indexes per table, by reference is *required*
  - Otherwise we would be replicating entire tuples
  - Q: Why is replicating a problem?
  - Replicating data leads to complexity during updates, so we want to avoid
    - Need to make sure that all copies of the data are kept in sync.

Connecting Back to the Storage Layer

- So far, we have been talking about a B+tree index pointing to unordered pages in a heap file
- This is not the only approach we can take.
- We'll talk about various alternatives for the:
  - Leaf nodes (the interface between index and the data)
  - Heap file (the actual data, if outside the index) ⬅ *this is next*

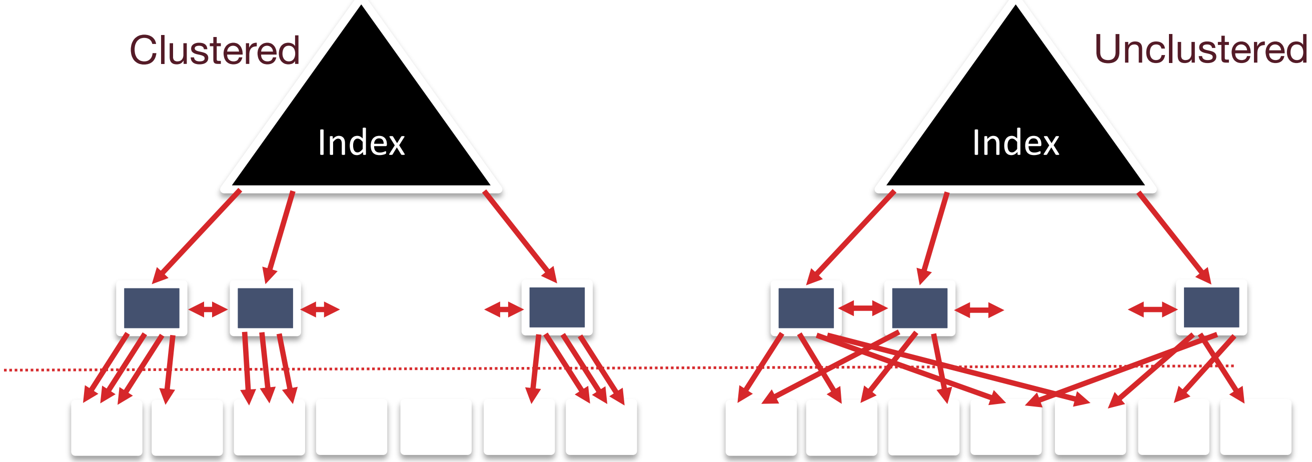
Clustered vs. Unclustered Index

- By-reference indexes (Alt 2 and 3) can be *clustered* or *unclustered*
    - In reality, this is a property of the heap file associated with the index!
- Clustered index:
    - Heap file records are kept *mostly* ordered according to **search keys** in index
        - Heap file order need not be perfect: this is just a performance hint
        - As we will see, cost of retrieving data records through index varies greatly based on whether index is clustered or not!

- Note: different definition of “clustering” in AI/data mining:
    - grouping nearby items in a high-dimensional space or network

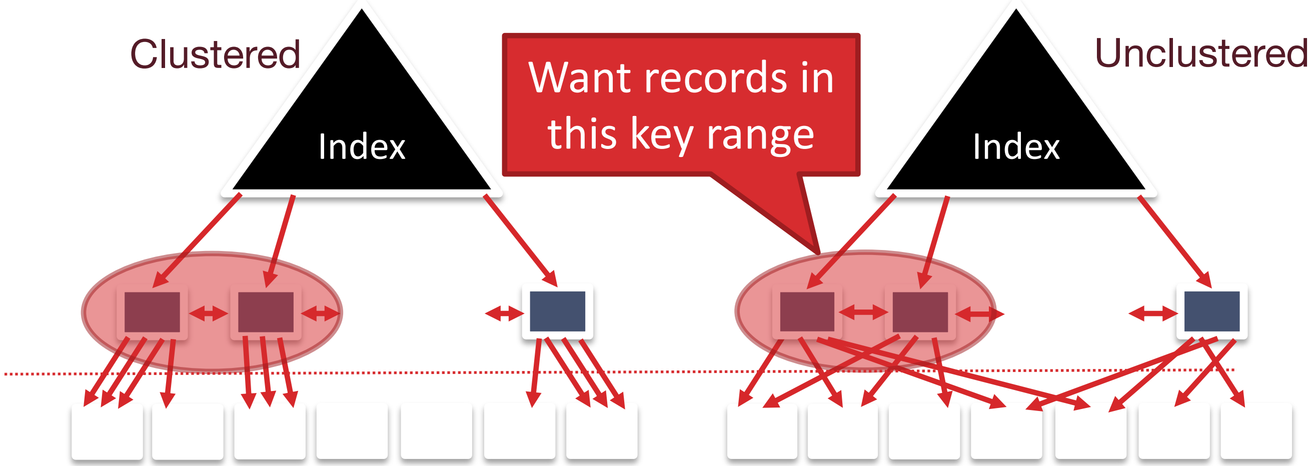
Clustered vs. Unclustered Index Visualization 1

- To build a clustered index, first sort the heap file
    - Leave some free space on each block for future inserts
    - We then try to respect this order "as much as possible"
- In an unclustered index, there is no such restriction



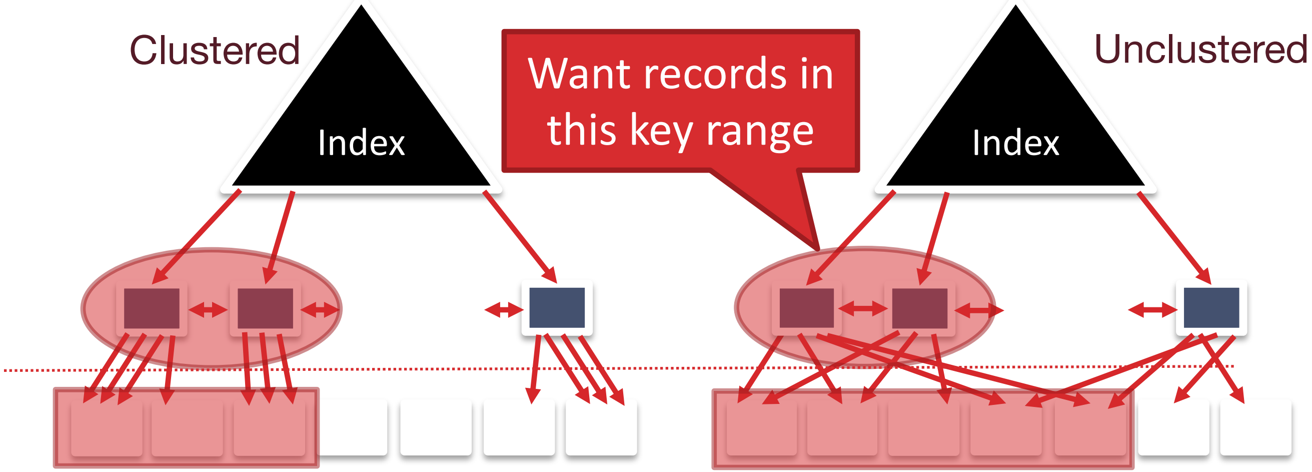
Clustered vs. Unclustered Index Visualization 2

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - We then try to respect this order “as much as possible”
- In an unclustered index, there is no such restriction



Clustered vs. Unclustered Index Visualization 3

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - We then try to respect this order “as much as possible”
- In an unclustered index, there is no such restriction

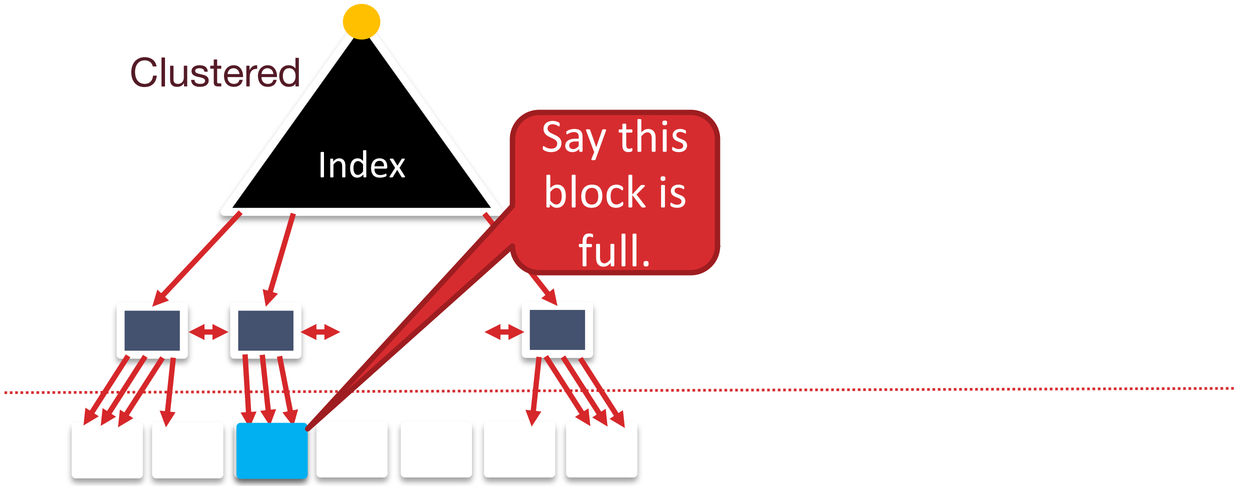


3 Heap file pages vs. 5 pages.

In general unclustered can be arbitrarily bad!

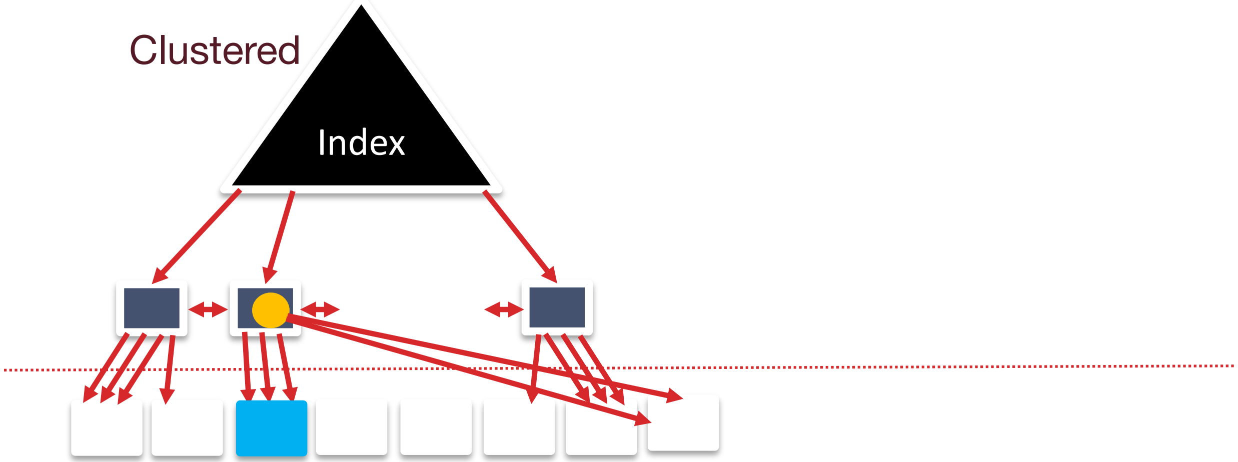
Clustered vs. Unclustered Index Visualization 5

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - We then try to respect this order "as much as possible"
- Blocks at end of file may be needed for inserts
  - Order of data records is "close to", but not identical to, the sort order



Clustered vs. Unclustered Index Visualization 6

- To build a clustered index, first sort the heap file
  - Leave some free space on each block for future inserts
  - We then try to respect this order “as much as possible”
- Blocks at end of file may be needed for inserts
  - Order of data records is “close to”, but not identical to, the sort order



Clustered vs. Unclustered Indexes Pros

- Clustered Index Pros
  - Efficient for range searches due to potential locality benefits
    - Sequential disk access, prefetching, etc.
  - Support certain types of compression
    - More soon on this topic

Clustered vs. Unclustered Indexes Cons

- Clustered Cons
    - More expensive to maintain
        - If we don’t maintain, ends up becoming closer to unclustered after many inserts
        - To maintain, we need to periodically update heap file order
            - Can be done on the fly (more expensive per update, but lookup perf is good throughout)
            - Or lazily (less expensive per update but performance can degrade)
    - To reduce cost of maintenance, heap file usually only **packed to 2/3** (or some other fraction <1) to accommodate inserts