Indexing 4

Instructor: Beom Heyn Kim

beomheynkim@hanyang.ac.kr

Department of Computer Science

Overview

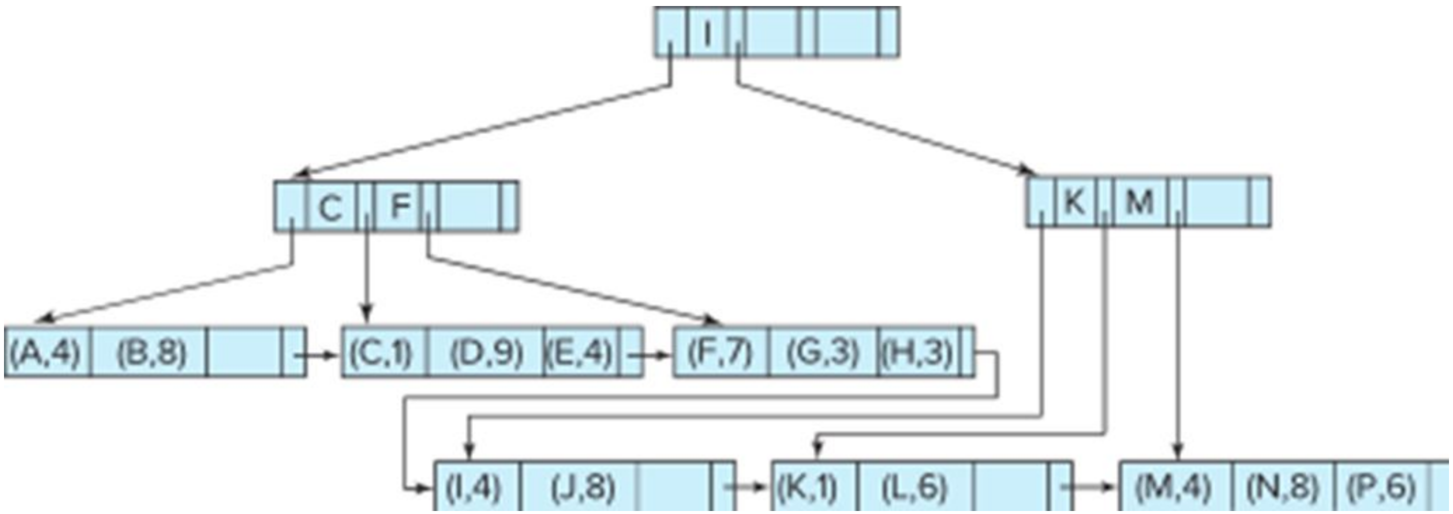
- B+-Tree Extensions
- Assignments

B+-Tree File Organization

- B^+-Tree File Organization:
  - Leaf nodes in a B^+-tree file organization store records, instead of pointers
  - Helps in keeping data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B+-Tree File Organization (Cont.)

- Example of B+-tree File Organization



- Good space utilization is important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
- Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

Secondary Indices and Record Relocation

- B+-tree file organization may change the location of records even when the records have not been updated
  - E.g. when a leaf node is split, many records are moved to a new node
- If a record moves, all secondary indices that store record pointers have to be updated
  - Node splits in B^+-tree file organizations become very expensive
- *Solution*: use search key of B^+-tree file organization instead of record pointer in secondary index
  - Add record-id if B^+-tree file organization search key is non-unique
  - Extra traversal of file organization is needed to locate record
    - Higher cost for queries, but node splits are cheap as we no longer need to update secondary indices

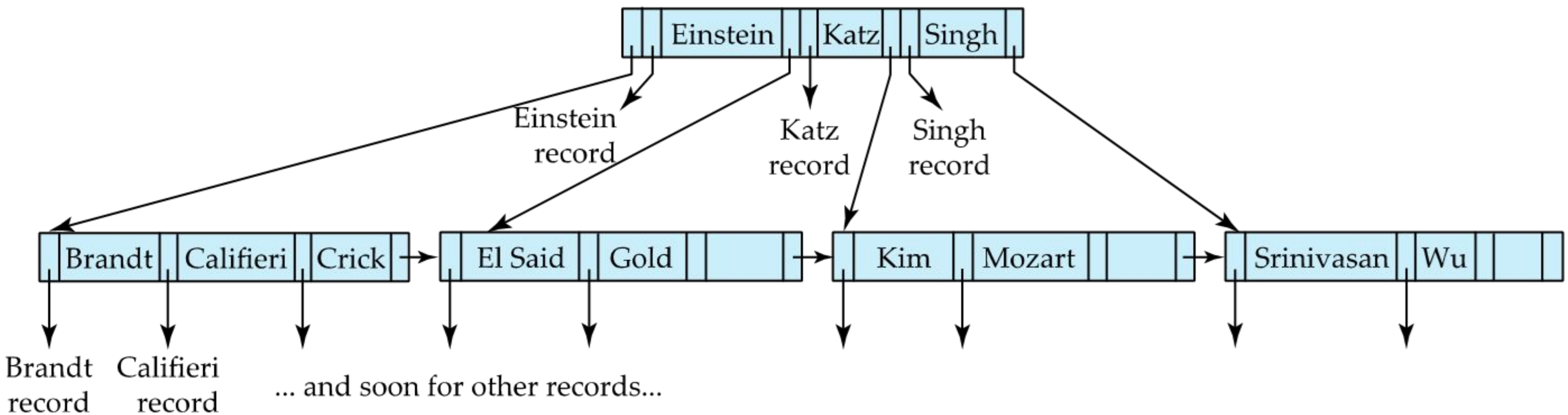
Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization (fraction of the free space) as criterion for splitting/merging, not number of pointers
- Strings can be long
  - smaller fanout and increased tree height
    - Slower lookup
- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g., “Silas” and “Silberschatz” can be separated by “Silb”

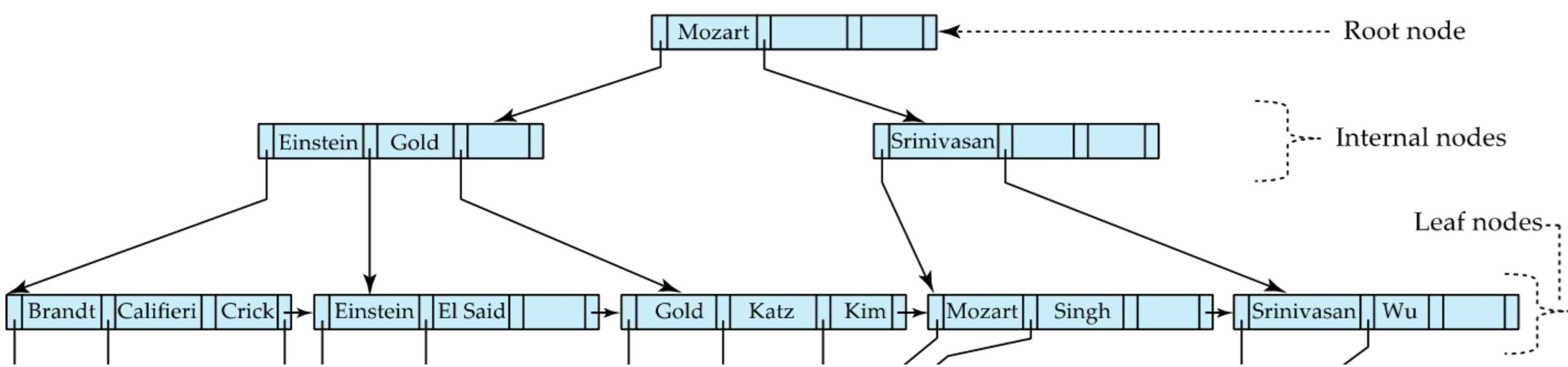
Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B^+-tree requires ≥ 1 IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (**bulk loading**)
- Efficient alternative 1:
  - sort entries first (using efficient external sort-merge algorithms discussed in Section 15.4)
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: **Bottom-up B^+-tree construction**
  - As before sort entries
  - And then create tree layer-by-layer, starting with leaf level
  - Implemented as part of bulk-load utility by most database systems

B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



Indexing on Flash

- Random I/O cost much lower on flash compared to magnetic disks
  - 20 to 100 microseconds for read/write compared to 5 to 10 milliseconds
- Writes are not in-place, and (eventually) require a more expensive erase
- The optimum B+-tree node size for flash is smaller than that with magnetic disk, since flash page size is smaller than the disk block size
  - Reduce the tree-node sizes to match to flash pages
- Bottom-up B+tree construction provides significant performance benefits, since it minimizes page erases
- Several extensions and alternatives to B+-tree have been proposed to reduce the number of erase operations

Indexing in Main Memory

- B+-tree indexing can be used to index in-memory data
- Some optimizations are possible
    - Reducing space overhead is important
        - redistributing using more than 1 sibling like we saw earlier can be helpful
    - Cache misses are expensive, so B+-trees with small nodes that fit in cache line are preferable
    - For databases with large data not fit entirely in memory, we can use large node size to optimize disk access, but structure data within a node using a tree with small node size, instead of using an array, to reduce cache misses.

Overview

- B+-Tree Extensions
- Assignments

Assignments

- Reading: Ch14.4
- Practice Excercises: 14.9

Solutions to the Practice Excercises:
https://www.db-book.com/Practice-Exercises/index-solu.html

The End