Indexing: Part V

CPS 216
Advanced Database Systems

Announcement

- Homework #2 due today (February 26)
  - Clarification on linear hashing capacity
- Midterm next Monday (March 3)
  - Everything up to (including) Monday's lecture
  - Open-book, open-notes
- No class next Wednesday (March 5)
- Course project proposal due in 7 days (March 5)
  - By email to `junyang@cs.duke.edu`
- Recitation session this Friday
  - Homework #2 sample solution
  - Midterm review

MMDB

- Traditional DBMS
  - Data resides on disk
  - Data may be cached in main memory for access
- Main-memory database system (MMDB)
  - Memory capacity doubles every 18 months
  - Many databases can now fit in main memory
  - Data permanently resides in main memory
  - Backup on disk

Disk versus main-memory indexing

4

- Primary goals for disk-oriented index design
- Primary goals for main-memory index design
- Design choices revisited
  - Make each index node fit on exactly one block?
  - Make fan-out as large as possible?
  - Store index key values in the index?

Classic index structures

5

- Arrays (a.k.a. "inverted" tables)
  - A list of tuple pointers, sorted by the index key
  - Pros:
  - Cons:
- AVL trees
  - Binary search tree balanced by rotations
  - Pros:
  - Cons:

Classic index structures (cont'd)

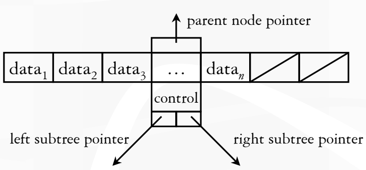
6

- B-trees (why not B^+-trees for main memory?)
  - Use a smaller index node size to avoid waste in space
  - Pros:
- Hash-based indexing
  - Pros:
  - Cons:

T-tree

- A balanced binary tree (like AVL)
- Many elements in each node; nodes do not need to be full (like B-tree)
- Rebalancing is done using rotations (like AVL, but much less frequently)
- Much data movement happens within a single node (like B-tree)

T-tree node



- $data_1$, $data_2$, …, $data_n$ are sorted (they can be pointers to actual records)
- Not all entries need to be occupied (significantly reducing reorganization cost)
- Everything found in the left subtree < $data_1$
- Everything found in the right subtree > $data_n$
- Heights of left and right subtrees differ at most by 1

Insert

Insert x

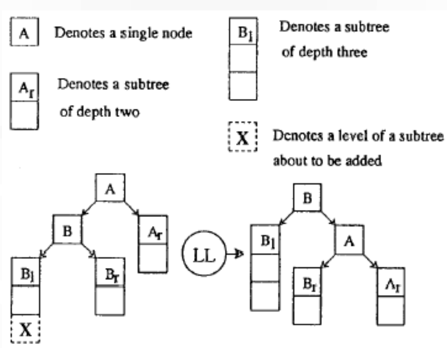
- Search for the "bounding" node such that $data_1 < x < data_n$
  - If the node has enough space, insert x here
  - Otherwise, remove $data_1$ from the node and insert it into the rightmost node in the left subtree
- If search exhausts the tree and no bounding node is found
  - Insert x into the last node on the search path if the node has enough space
  - Otherwise, create a new leaf with x
- Balance the tree if necessary when a new leaf is created

Delete

- ❖ Search for the element and remove it
- ❖ If the node underflows, borrow the smallest value from the leftmost node of the right subtree
- ❖ If the node is a half leaf (one subtree is empty and the other is a leaf), merge the leaf into it if possible
- ❖ If the node is empty, delete it and balance the tree if necessary

☞ Note: T-tree leaf nodes can be nearly empty

Example rotation for tree balancing



Experiment results

- ❖ Keep in mind these results were for 1986 systems…
  - ▪ CPU/memory speed gap was not as large back then
- ☞ Binary search is expensive because of address calculation
- ☞ Following stored pointers is faster
- ❖ Array
- ❖ AVL
- ❖ B-tree
- ❖ T-tree

Cache-sensitive main-memory indexing

- CPU speed doubles every 18 months
- Memory performance merely grows 10% per year
- ☞ Cache behavior becomes crucial for main-memory indexes
- ☞ Store search key values back inside indexes again!

Index structures revisited

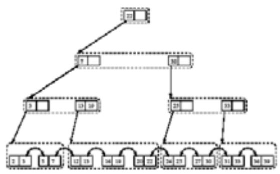
- Array
- T-tree
- B^+-tree
  - Make a node fit in a cache line
  - Overall misses: $\log_m n$, where m is the number of keys per node, and n is the total number of keys
  - ☞ Back to the old game: make m as large as possible for a cache line!

CSS-tree (*VLDB* 1999)

- Cache-sensitive search tree
- Similar to B^+-tree
- Eliminate child pointers to make space for more keys (thus larger m)
  - Assume fixed-size table and fan-out (like ISAM)
  - Nodes are stored level by level from left to right
  - Position of a child can be calculated
- ☞ Disadvantage:

CSB^{+}-tree (*SIGMOD* 2000)

- Start with a CSS-tree and add some pointers back to deal with updates
  - For each node, put its all child nodes into a node group
    - Within a node group, nodes are stored consecutively
  - Only a pointer to the node group is needed
- Example: a CSB^{+}-tree of a maximum fan-out of 2



Conclusion

- Things change
  - T-tree
    - CPU was still slow: address calculation was expensive
    - Ditched calculated addresses in favor of stored pointers
  - CSS-, CSB^{+}-trees
    - CPU and cache are now much, much faster than memory
    - Ditched stored pointers in favor of calculated addresses
- Then they don't
  - It is all about optimizing for speed gaps at various levels of storage hierarchy
    - Cache vs. memory, memory vs. disk