Database Management Systems

Indexing

Tree Index (I)

- Binary Search Tree: too many levels (e.g., too many internal blocks to be read), may become unbalanced
- Multiway Search Tree: A m-way search tree is a tree in which
  - The nodes hold between 1 to m-1 distinct keys
  - The keys in each node are sorted
  - A node with k values has k+1 subtrees, where the subtrees may be empty. The i'th subtree of a node [v1, ..., vk], 0 < i < k, may hold only values v in the range vi<v<vi+1(v0 is assumed to equal -Inf, and vk+1 is assumed to equal Inf).
  - A m-way tree of height h has between h and m^h – 1 keys.

Tree Index (II)

- (a,b)-tree: A multiway search tree with the restrictions that all leaves are at the same depth and all internal nodes have between a and b children, where a and b are integers such that 2 <= a <= (b+1)/2. The root may have as few as 2 children.

- B Tree: is an (a,b) tree.
  Definition: A balanced search tree in which every node has between ceiling(d/2) and d children, where d > 1 is a fixed integer. d is the order. The root may have as few as 2 children. This is a good structure if much of the tree is in slow memory (disk), since the height, and hence the number of accesses, can be kept small, say one or two, by picking a large d.

- B+ Tree:
  Leaf nodes and internal nodes have different structures. Internal nodes only store search keys and child pointers. Leaf nodes stores the actual data items.

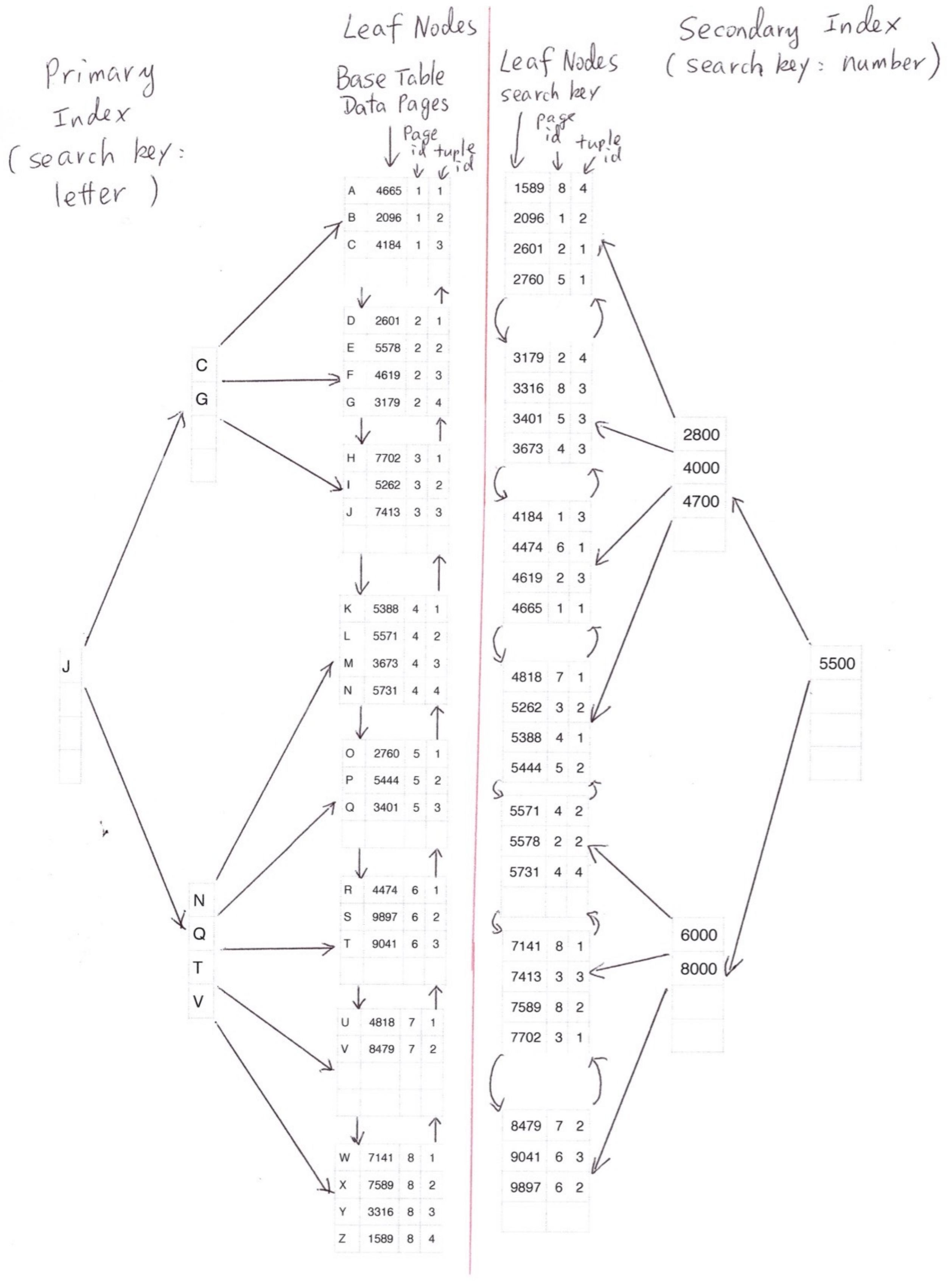
B+ Tree

- Index Blocks (internal nodes)
    - Each block stores at most m values (search keys) and m+1 pointers
    - Each block stores at least floor(m/2) values and floor(m/2)+1 pointers
    - Except the root node. The root node should have at most m values and m+1 pointers, and at least 1 value and 2 pointers.
- Leaf Blocks
    - Each block stores at most n records
    - Each block stores at least ceiling(n/2) records
    - Each block also contains two pointers pointing to the previous and next block
    - Records can be tuples themselves, or they may be key values plus tuple identifiers (like pointers, but for secondary storage)

B+ Tree Index Types

- Sparse (primary, clustered) index
- Dense (secondary, non-clustered) index

Example



A	4665	1	1
B	2096	1	2
C	4184	1	3
D	2601	2	1
E	5578	2	2
F	4619	2	3
G	3179	2	4
H	7702	3	1
I	5262	3	2
J	7413	3	3
K	5388	4	1
L	5571	4	2
M	3673	4	3
N	5731	4	4
O	2760	5	1
P	5444	5	2
Q	3401	5	3
R	4474	6	1
S	9897	6	2
T	9041	6	3
U	4818	7	1
V	8479	7	2
W	7141	8	1
X	7589	8	2
Y	3316	8	3
Z	1589	8	4

1589	8	4
2096	1	2
2601	2	1
2760	5	1
3179	2	4
3316	8	3
3401	5	3
3673	4	3
4184	1	3
4474	6	1
4619	2	3
4665	1	1
4818	7	1
5262	3	2
5388	4	1
5444	5	2
5571	4	2
5578	2	2
5731	4	4
7141	8	1
7413	3	3
7589	8	2
7702	3	1
8479	7	2
9041	6	3
9897	6	2

Using B+ tree index

- Look up
  - key search (select * from T where number = 5578;)
  - range search (select * from T where letter >= ‘K’ and letter <= ‘P’;)
- insertion (overflow)
  - always try to insert into a leaf node. If there is room, done.
  - If there is no room in the proper leaf, we split the leaf into two and divide the keys between the two new nodes, so each is half full or just over half full.
  - The splitting of nodes at one level appears to the level above as if a new key- pointer pair needs to be inserted at that higher level. So the insert can be done recursively up the path.
  - It may be possible to split the root.
- deletion (underflow)
  - remove the victim
  - when underflow happens, try first to steal an element from one of the sibling nodes. If neither the sibling nodes has a spare element, we need to merge two nodes, and the delete would happen recursively up the path.
  - Some B+ tree implementations don’t fix up underflow nodes. The rationale is that most tables will soon grow again and fill the space in no time.

Hash Table Indexing

- Main memory hash table
- Disk hash table index: each bucket is a block (or a page)
- Static one: h(k) = d, go to block d (or page d). There may be overflow pages linked in a list.
- flaw with static one: the overflow list may grow very long.

Dynamic hash table indexing

- It’s an extensible hash table.
- The bucket array is a growing array. (The size of the array doubles when needed.)
- d = h(k) has many bits.
- start with level = 0 and size of the array = 2, and only use the first bit of d.
- when array size doubles, increase the level by one. Then use level+1 bits of d to visit the page.