Tree Based Indexes

• Want to guarantee worst case time $O(\log n)$ for dictionary operations on ordered data
  – Primarily want to work with large amounts of data
  – Also Want to support range queries efficiently
• The data structure is called an index because
  – Data is stored with various ordered attributes (such as age, salary, employee identification number)
  – The range query may come on any of the attributes
  – We do not want to duplicate the data
Storing Large Amounts of Data

- Store choice for large amounts of data is disk, not RAM or tape
- Why not store in main memory?
  - Costs too much. Rs. 4000 will buy you either 256MB of RAM or 40GB of data
  - Volatile.
- Reading or writing data on disk is slow (about 8ms).
- Data is stored and retrieved in units called disk blocks or pages
- Note that in fact, we have a memory hierarchy for even slower data access (tapes for archival purposes)
Binary Trees May Not Suffice

• Assume we have 10 million items, each key is a name occupying 32 bytes, and a record is 256 bytes
  – Average successful search takes $1.38 \lg 10^7 \approx 1.38 \lg 2^{21} \approx 32$ disk accesses
  – In a time shared environment with 20 users, this might take about 5 seconds
  – Worse, a few nodes might be three times deeper and we need 100 disk accesses or about 16 seconds
• So searching 4 records would take about a minute
• We might be willing to write complicate code to get better performance (e.g., 3 disk accesses instead of 32 or 100)
Multi-way Branches Are Sometimes Useful

- We get $\log_m n$ performance for $n$ nodes using $m$-way branches
- A node now has $m - 1$ pieces of data and $m$ pointers
  - If page size is $8192 = 2^{13}$ bytes a block contains $32(m - 1)$ keys.
  - We store $m$ pointers to other blocks, and each pointer is 4 bytes. Thus total storage is $36m - 32 = 8192$ or $m = 228$
  - An $m$-ary tree of height 3 has $228^3 \geq 10$ million nodes
  - The search time is about 0.47 second.
  - And we get other pieces of data in our buffer!
- 32 has reduced to 3, but we probably *don’t* want to write complicated code to make 3 main memory accesses instead of 100 (memory access time is in nanoseconds)
- But maybe we cannot afford enough RAM
The DBMS Provides Many Abstractions

• For several reasons, a programmer doesn’t deal directly with disks but is provided various abstractions by the database management system (DBMS)
  – The time to access a page depends on its location on disk. Careful placement of pages on the disk to exploit the geometry can minimize access time
  – The illusion of several disks (in an array) posing as one disk can be provided

• The buffer manager brings pages into RAM

• Database pages are organized into files (not to be confused with operating system files), and higher level DBMS code view the data as a collection of records with record id (rid) pointing to physical disk pages
Files of Records

- So instead of asking for pages, the programmers asks to
  - Insert/delete/modify records
  - Read a particular record (using rid)
  - Scan a subset of records (with some conditions on the search)
- The records are in one to one correspondence with data, and so do not fit in main memory either
- File structures include
  - Unordered, linked list like or sorted files
  - Or hashed files (a directory like structure)
- An index is an auxiliary structure to the basic file structure that is value-based (rather than rid based)
Indexes

- An index contains a collection of data entries and supports efficient retrieval of all data entries with a given key value $k$.
- A data entry consists of the key $k*$ and enough information to retrieve the physical record that contains data matching the key $k$.
- So for an employee database
  - We might have an index on salary. The index (stored on disk) will now contain pointers to relevant pages that will support efficient access of salary information.
  - We might have an index on age. The index file now contains pointers to support efficient access of age related information.
- As mentioned earlier, if data entries are stored in sorted order, binary search can be performed, but this is costly.
- Stores data entry pages *and*, in addition, index pages in a tree-like fashion is more efficient.
The Basic Abstraction of B+ trees

• So just like non-linear main memory data structures such as AVL trees, we have B+ trees

• The basic abstraction we care about for an object $x$
  
  – If the object is in main memory, we can refer to the fields of the object
  
  – Otherwise, we need to perform a DiskRead($x$) to fetch the data
  
  – We write the data back to disk using DiskWrite($x$)

• We do not care that underneath the hood, the file manager and the buffer manager are called to manage physical pages on disk

• We want to minimize these disk accesses.

• We prefer a one-pass algorithm instead of the usual top-down bottom up passes (such as in the rebalancing method of AVL trees)
B+-Tree Node Structure

- Rooted tree where every node $x$ has $n[x]$ the number of keys currently stored in $x$, the $n[x]$ keys themselves stored in nondecreasing order so that keys $k_1[x] \leq k_2[x] \leq \ldots k_{n[x]}[x]$, and a boolean value leaf[$x$] to indicate whether $x$ is a leaf.

- Each internal node also contains $n[x] + 1$ child pointers $c_i[x]$

- Leaf nodes are data entry nodes and point to the actual data.

- The keys $k_i[x]$ separate the range of keys stored in each subtree: If $p_i$ is any key stored in the subtree with root $c_i[x]$, then $p_1 < k_1[x] \leq p_2 < k_2[x] \leq \ldots < k_{n[x]}[x] \leq p_{n[x]+1}$
B+-Tree Constraints

- All leaf nodes are at the same depth and chained together in a doubly linked list
- For $t \geq 2$
  - Each node other than the root must have at least $t - 1$ keys.
  - Every node can contain at most $2t - 1$ keys.
  - The root node has 1 to $2t - 1$ keys (unless the tree is empty)
- Some things to note
  - Nodes are fixed size
  - The occupancy factor of our tree is $\frac{t-1}{2t-1}$, which is in $[0.33, 0.5)$ (implementation decision)
  - Our leaves are the same size as internal nodes, but this need not be the case.
  - Most of the time, the root node is cached in main memory, and is not a leaf.
Height of a B+ Tree

• Although actual data are only in the external nodes, let us, for the sake of analysis, count keys in internal nodes as well.

• Let \( n(H) \) be a tree of height \( H \) with minimum number of keys:
  - The root has only two children \( L \) and \( R \) each of height \( h = H - 1 \)
  - The number of keys in \( L \) must be at least \( \frac{t^{h+1}-1}{t-1} \)

\[
\begin{align*}
  n &\geq 1 + 2\frac{t^{h+1}-1}{t-1} \\
  \frac{n-1}{2} &\geq \frac{t^{h+1}-1}{(t-1)} \geq \frac{t^{h+1}-1}{t} \geq t^h - \frac{1}{t} \\
  \log_t \frac{n-1}{2} &\geq h + 1
\end{align*}
\]

• \( h \in O(\log_t n) \), so \( H \in O(\log_t N) \) where \( N \) is the number of keys in external nodes.

• Typical order \( t = 100 \), average fanout 133. If the height is 4, 312,900,700 records can be stored.
Example B+ Tree

B+-tree with the parameter $t = 2$

- Data is stored only in external nodes. This is very natural given the DBMS abstraction.
- Storing data in internal nodes implies storing pointers to pages in internal nodes, thereby decreasing the branching factor.
- Doubly Linked List (not always shown) enables efficient range retrievals (Awkward if data is in internal nodes as well)
Pseudo Code For Search

- Search begins at root, and key comparisons direct to leaf
- Within a node, any appropriate search method may be used
- Invoked using search(root[T], x);
- Number of disk I/O proportional to height of tree

```c
page search (Node x, Key z) {
    for (i = 1; i <= n[x] && z >= k[i]; ++i);
    if (leaf(x))
        if (k[i] == z) return c[i];
        else return nil;
    y = DiskRead(c[i]);
    return search(y, k);
}
```
Inserting a Data Entry

- Find correct leaf $L$ and put data entry in $L$
- If $L$ has enough space, done!
- Else must split $L$ into $L$ and a new node $L_2$
  - Redistribute entries evenly, and copy middle key
  - Insert index entry pointing to $L_2$

![Diagram showing data insertion process with examples.]
Inserting a Data Entry

- If index page gets full, we need to *push up* middle key
- Split may ‘grow’ tree bottom up (When the root splits, height increases)
- Note: Factor of two saving if we code in top down one pass

![Diagram showing insertion process]

- Note the difference between copy and pull (copy and push for bottom up)
Insert a Data Entry

• Splitting may be avoided by redistributing keys; this is usually not done in practice

• Minimum occupancy is guaranteed in both leaf and index page splits
Pseudo Code for Split

Assumption: \( x \) is not full. Child \( y = c[i] \) is full

\[
\text{b-tree-split-child}(x, i, y) \{
\text{new}(L2); \text{leaf}(L2) = \text{leaf}(y);
\text{if} (! \text{leaf}[y]) \{ \quad \text{// first fix L2}
\quad n[z] = t - 1;
\quad \text{for} \ (j=1; j < t-1; ++j) \quad k[j][L2] = k[j+1][y];
\quad \text{for} \ (j=1; j < t; ++j) \quad c[j][L2] = c[j+1][y];
\}
\text{else} \{ \quad \text{// omitted}
\quad n[y] = t - 1; \quad \text{// next fix y}
\quad \text{// and finally fix x}
\quad \text{for} \ (j = n[x]+1; j > i; --j) \quad c[j+1] = c[j];
\quad c[i+1] = L2;
\quad \text{for} \ (j = n[x]; j >= i; --j) \quad k[j+1] = k[j];
\quad k[i][x] = k[t][y]; \quad n[x] = n[x] + 1;
\quad \text{DiskWrite}(L2, y, x);
\}
\]
Pseudo Code for Insert

Basic Idea: Split a child to guarantee that the recursion never descends to a full node

\[
\text{b-tree-insert}(\text{Tree } T, \text{ Key } p) \{ \\
\text{r} = \text{root}[T]; \\
\text{if} \ (n[r] \neq 2t - 1) \text{return insert-nonfull}(r, p); \\
\text{new} \ (S); \\
\text{root}[T] = s; \ \text{leaf}[s] = \text{false}; \\
\text{n}[s] = 0; \ \text{c}[1] = r; \\
\text{b-tree-split-child}(s, 1, r); \\
\text{insert-nonfull}(s, p); \\
\}
\]
Pseudo Code for Insert

\[
\text{insert}-\text{nonfull}(\text{Node } q, \text{ Key } p) \{
\]
\[
i = n[q];
\]
\[
\text{if} \ (\text{leaf } (q)) \{
\]
\[
\text{while} \ (i > 0 \&\& p < k[i]) \{
\]
\[
k[i+1] = k[i]; --i;
\]
\[
}\]
\[
k[i+1] = p; n[q] = n[q] + 1; \text{DiskWrite}(q);
\]
\[
\} \]
\[
\text{else} \{
\]
\[
\text{while} \ (i > 0 \&\& p < k[i]) --i;
\]
\[
i++;
\]
\[
y = \text{DiskRead}(c[i]);
\]
\[
\text{if} \ (n[y] = 2t-1) \{
\]
\[
b-tree-split-child(q,i,y);
\]
\[
\text{if} \ (p > k[q]) ++i;
\]
\[
}\]
\[
\text{insert}-\text{nonfull}(y, p);
\]
\[
\}
\]
Removing a Data Entry

- Start at root, find leaf $L$ where entry belongs
- Remove the entry
  - If $L$ has $t - 1$ keys after remove, done
  - If $L$ has less keys, try to redistribute borrowing from sibling node (may not always be possible)

- Key is \textit{copied} and evenly distributed. The exact way this is performed depends on the implementation.
Removing: The Basic Merge Idea

- If re-distribution fails, merge the leaf $L$ with one of its siblings
- Must delete entry (pointing to $L$ or sibling) from parent of $L$
- Index entry is tossed away!
- This implies that parent could violate B+-tree constraint
- And merge could propagate to root, decreasing height
Removing: The Cascade Case

- Anticipated merge enables a one pass top down algorithm
- Observe push down of index entry (rather than the toss)
Pseudo Code for Remove

• Basic Idea: Guarantee that the recursion never descends to a node that has less than \( t \) keys (not \( t - 1 \) keys)

• Structure code so that when \( \text{remove}(x, p) \) is called, \( x \) has \( t \) keys or more.
  – To make this happen, first try redistributing, and
  – If that fails, merge
  – While merging, push down (if going from parent to internal node. Otherwise toss away
  – Root is a special case
Pseudo Code For Remove

// x is guaranteed to be at least half full
remove-halfFull(x, p) {
    if (leaf(x)) return remove-leaf(x, p);
    i = n[x];
    while (i > 0 && p < k[i]) i--;
    ++i;
    z = DiskRead(c[i]);
    if (n[z] >= t) return remove-halfFull(z, p);
    // try to borrow, or worse, merge
    redistribute(x, i, z);
    // recurse, now z is guaranteed to be half full
    remove-halfFull(z, p);
}
Pseudo Code for Remove

// x is the parent, z=c[i][x] is sparse
redistribute(x, i, z) {
  if (i > 1) {
    // Look for a donor on the left
    z' = DiskRead (c[i-1][x]);
    if (n[z'] >= t) return rotateL (x, i-1, z);
    // rotate involves DiskWrite of x, z and left sibling of z
  }
  if (i <= n[x]) {
    z' = DiskRead (c[i+1]);
    if (n[z'] >= t) return rotateR (x, i);
  }
  return merge (x, i, z)
}
Root Special Case

remove (T, p) {
    r = root [T];
    // easy case
    if (n[r] >= 2) return remove-halfFull(r, p);
    // second easy case: both children are sparse
    L = DiskRead(c[1]); R = DiskRead(c[2]);
    if (n[L] == t−1 && n[R] == t−1)
        merge(r, 1, L);
    // headed to right, and right might be sparse
    if (p >= k[1][r])
        if (n[R] == t−1) redistribute(r, 2, R);
    else
        // headed to left which might be sparse
        if (n[L] == t−1) redistribute(r, 1, L);
    // now r is either half full, or
    // it doesn’t matter, relevant child is half full
    return remove-halfFull(r, p);
}
B-trees, 2-3 trees, 2-3-4 trees versus B+-trees

- Note that if our ideas work, then we can retro-fit the disk based tree algorithm for a logarithmic main memory algorithm.
  - A disk based algorithm stores all data entries in leaves resulting in a B+-tree
  - A main memory algorithm does not need to distinguish between nodes resulting in data at any level
  - 2-3-4 trees, 2-3, (a-b) trees are special cases of B-trees

- Choose whether the increase in coding complexity is worth it