Indexing with B-trees

Anastassia Ailamaki
http://www.cs.cmu.edu/~natassa

Problem

Given a large collection of records,

find similar/interesting things,

i.e.,

allow fast, approximate queries

Indexing

- primary key indexing
  - B-trees and variants
  - (static) hashing
  - extendible hashing
- secondary key indexing
- spatial access methods
- text
- ...

Primary Key Indexing

- find employee with ssn=123

attributes in a record Rₙ

<table>
<thead>
<tr>
<th>R₁</th>
<th>R₂</th>
<th>R₃</th>
<th>R₄</th>
<th>R₅</th>
<th>R₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>a₂</td>
<td>a₃</td>
<td>a₄</td>
<td>a₅</td>
<td>a₆</td>
</tr>
</tbody>
</table>

table R

sequential scan

using an index

Balanced “n-way” search trees

B-trees

- **Most successful** family of index schemes
  - B-trees
  - B⁺-trees
  - B⁻-trees
  - Can be used for
    - primary/secondary, or
    - clustering/non-clustering index.
  - Balanced “n-way” search trees

B-trees: Example

Here is a B-tree of order 3:

<6

6

9

>9

1

2

3

4

5

6

7

8

9

10

11
B-tree Properties

In a B-tree of order n:
- key order preserved
- at most n pointers
- at least n/2 pointers (except root)
- all leaves at the same level
- if number of pointers is k, node has exactly k-1 keys
- (leaves are empty)

\[ p_1 \quad p_2 \quad \ldots \quad p_n \]

B-tree Properties (cont.)

- “block aware” nodes: each node -> disk page
- \( O(\log(N)) \) for everything! (ins/del/search)
- typically, if \( m = 50 - 100 \), then 2 - 3 levels
- utilization \( \geq 50\% \), guaranteed; on average 69%

Exact-Match Queries

E.g., ssn=8

\[ \begin{array}{c}
1 \quad 3 \\
6 \quad 9 \quad 13 \\
\end{array} \]

\( U \) steps (= disk accesses)

Range Queries

E.g., 5<salary<8

\[ \begin{array}{c}
1 \quad 3 \\
6 \quad 9 \quad 13 \\
\end{array} \]

Proximity Queries

E.g., nearest neighbor searches: salary ~ 8

B-trees: Insertion

- Insert in leaf; on overflow, push middle up (recursively)
- split: preserves B-tree properties
B-trees: Insertion (cont.)

Easy case: Tree T0; insert ‘8’

Hardest case: Tree T0; insert ‘2’

Hardest case: Tree T0; insert ‘2’

Overflow; push middle

push middle up

Final state
B-trees - insertion

- Q: What if there are two middles? (eg, order 4)
- A: either one is fine

Algorithm: Insertion of Key ‘K’

find the correct leaf node ‘L’;
if ( ‘L’ overflows ) {
    split ‘L’ by pushing middle key up to parent ‘P’;
    if (‘P’ overflows) {
        repeat the split recursively;
    } else {
        add key ‘K’ in node ‘L’; // maintain key order in ‘L’
    }
}

B-trees: Insertion Sketch

Algorithm:
1. insert in leaf
2. on overflow:
   push middle up (recursively – ‘propagate split’)

- Split preserves all B-tree properties (!!!)
- Notice how it grows:
  height increases when root overflows & splits
- Automatic, incremental re-organization

B-trees: Deletion

Rough outline of algorithm:
- Delete key;
- on underflow, may need to merge

In practice, some implementors just allow underflows to happen...

B-trees: Deletion Cases

- Case 1
  delete a key at a leaf – no underflow
- Case 2
  delete non-leaf key – no underflow
- Case 3
  delete leaf-key; underflow, and ‘rich sibling’
- Case 4
  delete leaf-key; underflow, and ‘poor sibling’
Easiest case: Tree T0; delete ‘3’

Case 2: delete a key at a non-leaf
- no underflow (e.g., delete 6 from T0)

Delete & promote, i.e.:

Case 2: delete a key at a non-leaf – no underflow (e.g., delete 6 from T0)

Q: How to promote?
A: pick the largest key from the left sub-tree (or the smallest from the right sub-tree)

Observation: every deletion eventually becomes a deletion of a leaf key
B-trees: Deletion Cases (cont.)

- Case 1: delete a key at a leaf – no underflow
- Case 2: delete non-leaf key – no underflow
- Case 3: delete leaf-key; underflow, and ‘rich sibling’
- Case 4: delete leaf-key; underflow, and ‘poor sibling’

B-trees: Deletion Case 3

- Case 3: underflow & ‘rich sibling’ (e.g., delete 7 from T0)

Case 3: underflow & ‘rich sibling’ (e.g., delete 7 from T0)

Delete & borrow

Rich sibling

B-trees: Deletion Case 3 (cont.)

- Case 3: underflow & ‘rich sibling’
  - e.g., delete 7 from T0

Delete & borrow

Rich sibling

B-trees: Deletion Case 3 (cont.)

- Case 3: underflow & ‘rich sibling’
  - e.g., delete 7 from T0

Delete & borrow

Rich sibling

B-trees: Deletion Case 3 (cont.)

- Case 3: underflow & ‘rich sibling’
  - e.g., delete 7 from T0

Delete & borrow

Rich sibling

B-trees: Deletion Case 3 (cont.)

- Case 3: underflow & ‘rich sibling’
  - e.g., delete 7 from T0

Delete & borrow

Rich sibling

B-trees: Deletion Case 3 (cont.)

- Case 3: underflow & ‘rich sibling’
  - e.g., delete 7 from T0

Delete & borrow

Rich sibling
B-trees: Deletion Case 3 (cont.)

Case 3: underflow & ‘rich sibling’
- e.g., delete 7 from T0

Delete & borrow

B-trees: Deletion Case 3 (cont.)

Case 3: underflow & ‘rich sibling’
- e.g., delete 7 from T0

Delete & borrow
THROUGH the parent

B-trees: Deletion Cases (cont.)

- Case 1: delete a key at a leaf – no underflow
- Case 2: delete non-leaf key – no underflow
- Case 3: delete leaf-key; underflow, and ‘rich sibling’

⇒ Case 4: delete leaf-key; underflow, and ‘poor sibling’

B-trees: Deletion Case 4

Case 4: underflow & ‘poor sibling’
- e.g., delete 13 from T0

A: merge w/ ‘poor’ sibling

B-trees: Deletion Case 4 (cont.)

Case 4: underflow & ‘poor sibling’
- e.g., delete 13 from T0

A: merge w/ ‘poor’ sibling
B-trees: Deletion Case 4 (cont.)

Case 4: underflow & 'poor sibling'
  - eg., delete 13 from T0)

- Merge, by pulling a key from the parent
- exact reversal from insertion: 'split and push up', vs. 'merge and pull down'

ie.:

B-trees: Deletion Case 4 (cont.)

Case 4: underflow & 'poor sibling'
  - eg., delete 13 from T0)

- Q: What if the parent underflows?
- A: repeat recursively

Algorithm: Deletion of Key ‘K’

locate key ‘K’, in node ‘N’
if (‘N’ is a non-leaf node) {
  delete ‘K’ from ‘N’;
  find the immediately largest key ‘K1’;
  /* which is guaranteed to be on a leaf node ‘L’ */
  copy ‘K1’ in the old position of ‘K’;
  invoke DELETION on ‘K1’ from the leaf node ‘L’;
} else {
  /* ‘N’ is a leaf node */
  ...next slide...

Deletion of Key ‘K’ (cont.)

if (‘N’ underflows ){
  let ‘N1’ be the sibling of ‘N’;
  if( ‘N1’ is “rich”){ /* ie., N1 can lend us a key */
    borrow a key from ‘N1’ THROUGH parent node;
  } else {
    /* N1 is 1 key away from underflowing */
    MERGE: pull key from parent ‘P’, merge it
    with keys of ‘N’ and ‘N1’ into new node;
    if( ‘P’ underflows) { repeat recursively }
}

FINAL TREE

<6
1 3
8

>6
7 9
1 3
8
B-trees in Practice

In practice:
- no empty leaves;
- pointers to records

B-trees in Practice (cont.)

In practice:
- no empty leaves;
- pointers to records

B-trees in Practice (cont.)

In practice:
- leaf nodes: (v1, rp1, v2, rp2, … vn, rpn)
- Non-leaf nodes: (p1, v1, rp1, p2, v2, rp2, …)

B+ trees: Motivation

B-tree – print keys in sorted order:
**B+ trees: Motivation (cont.)**

B-tree needs back-tracking – how to avoid it?

![Diagram](image)

**Solution: B+ - trees**

- Facilitate sequential ops
- They string all leaf nodes together

AND

- Replicate keys from non-leaf nodes, to make sure every key appears at the leaf level

**B+ trees**

![Diagram](image)

**B+ trees: Insertion**

E.g., insert ‘2’

![Diagram](image)

**B*-trees: Motivation**

- Splits drop utilization to 50%
- May increase height
- How to avoid them?

**B*-trees: Deferred Split!**

Instead of splitting, LEND keys to sibling! (through PARENT, of course!)

![Diagram](image)
B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling! (through PARENT, of course!)

FINAL TREE

B*-trees: Advantages

- Tree becomes
  - Shorter,
  - More packed,
  - Faster
- Rare case: improve together
  - space utilization
  - speed
- BUT: What if sibling has no room for ‘lending’?

B*-trees: deferred split!

- BUT: What if sibling has no room for ‘lending’?
- 2-to-3 split
  1. get the keys from the sibling
  2. pool them with ours (and a key from the parent)
  3. split in 3
- Details: too messy (and even worse for deletion)

Conclusions

- Main ideas: recursive; block-aware; on overflow -> split; defer splits
- All B-tree variants have excellent, $O(\log N)$ worst-case performance for ins/del/search
- It’s the prevailing indexing method

Performance Aspects of B-trees

Two parameters matter:

- Height $H$ (maximum search path)
  
  $H = 1 + \left\lceil \log_{F^*} \left( \frac{N}{C^*} \right) \right\rceil$

  - $N$ is the number of tuples
  - $C^*$ is the average number of entries in a leaf node, and
  - $F^*$ is the average number of entries in an index node.

- Size $S$ (number of pages tree occupies)
  
  $S = \sum_{i=1}^{H} (F^*)^{i-1}$, $1 \leq i < H$

Reducing the Number of Leaf

- Increase page size (hard)
- Shorten data length (values, tuples, pointers)
- Is it worthwhile to change the tuples to TIDs?
- No – extra page accesses!
  
  From Gray&Reuter: $1.1 \leq \log_{F^*} X$ must hold
  i.e., average fan-out really small or tuples $> 1K$
Increasing the Fanout

- Compression
  - Prefix – store differences (suffixes)
  - Suffix – store prefixes
- Prefix compression: sequential scan
  - “anchor” keys

Lehman and Yao – CC on B-trees

- “safe” node: node with <2k entries
- “unsafe” node: node with =2k entries
- Simple CC won’t do. Why?

Example

Transaction 1:
read x;
look for 15;
get ptr to y;
Transaction 2:
read x; read y;
insert 9
split y into y+y’
ERROR!!!

Previous B-tree CC algorithms

- Samadi 1976
  - lock the whole subtree of affected node
- Bayer & Schkolnick 1977
  - parameters on degree/type of consistency required
  - writer-exclusion locks (readers may proceed) upper
  - exclusive locks on modified nodes
- Miller & Snyder 1978
  - pioneer and follower locks
  - locked region moves up the tree
  - no modifications

B_{link}-tree

- Node + P2k+1 – pointer to next node at the same level of tree
- Rightmost node’s B-link is NULL
- IDEA:
  - Splitting is implemented as
  - legal to have “left twin” and no parent

Advantages

- Allows for “temporary fix” until all pointers are added correctly
- Link pointers should be used infrequently
  - because splitting a node is a “special case”
- “Level traversal” comes for free as a side effect
Algorithms

- Search
  - No locks needed for reads
  - Just move right as well as down
- Insertions
  - Well-ordered locks
  - Use stack to remember ancestors
  - Split while preserving links
- Deletions
  - No underflows, no merging