**Problem**

Given a large collection of (multimedia) records, find similar/interesting things, ie:

- Allow fast, approximate queries, and
- Find rules/patterns

---

**Reading Material**

[Ramakrishnan & Gehrke, 3rd ed, ch. 10]
Outline

Goal: ‘Find similar / interesting things’
- Intro to DB
- Indexing - similarity search
- Data Mining

Indexing - Detailed outline

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

In even more detail:

- B – trees
  - B+ – trees, B*-trees
  - hashing
Primary key indexing

• find employee with ssn=123

B-trees

• the most successful family of index schemes (B-trees, B+-trees, B*-trees)
• Can be used for primary/secondary, clustering/non-clustering index.
• balanced “n-way” search trees

Citation


• Received the *2001 SIGMOD innovations* award
• among the most cited db publications
  • www.informatik.uni-trier.de/~ley/db/about/top.html
B-trees

Eg., B-tree of order 3:

B - tree properties:

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

Properties

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

• Algo for exact match query? (eg., ssn=8?)

Queries

• Algo for exact match query? (eg., ssn=8?)

Queries

• Algo for exact match query? (eg., ssn=8?)
Queries

- Algo for exact match query? (eg., ssn=8?)

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary ~ 8)
Queries

- what about range queries? (e.g., $5 < \text{salary} < 8$)
- Proximity/ nearest neighbor searches? (e.g., $\text{salary} \sim 8$)

B-trees: Insertion

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

Easy case: Tree T0; insert ‘8’

B-trees

Tree T0; insert ‘8’

B-trees

Hardest case: Tree T0; insert ‘2’
B-trees

Hardest case: Tree T0; insert ‘2’

push middle up

B-trees

Hardest case: Tree T0; insert ‘2’

Ovf; push middle

B-trees

Hardest case: Tree T0; insert ‘2’

Final state
B-trees: Insertion

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

B-trees: Insertion

• Insert in leaf; 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

Overview

• B – trees
  – Dfn, Search, insertion, deletion
• B+ – trees
• hashing
Deletion

Rough outline of algo:
• Delete key;
• on underflow, may need to merge

In practice, some implementors just allow underflows to happen…

B-trees – Deletion

Easiest case: Tree T0; delete ‘3’
B-trees – Deletion

Easiest case: Tree T₀; delete ‘3’

```
       <6
       6 9
  >6 <9
  7 13
```

B-trees – Deletion

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

```
       <6
       6 9
  >6 <9
  7 13
```

Delete & promote, ie:
B-trees – Deletion

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

Delete & promote, i.e:

FINAL TREE
B-trees – Deletion

Case 2: delete a key at a non-leaf – no underflow (eg., 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

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: underflow & ‘rich sibling’ (eg., delete 7 from T0)
B-trees – Deletion

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

Delete & borrow, ie:

Rich sibling <6

<6 <9

>9

<9

Delete & borrow, ie:

Rich sibling NO!!
B-trees – Deletion

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

Delete & borrow, i.e:

1 3
<6 9
>6 <9
13
6

1 3
6
<6 9
>6 <9
13
B-trees – Deletion

- 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’

**FINAL TREE**

Delete & borrow, through the parent

B-trees – Deletion

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

**FINAL TREE**

Delete & borrow, through the parent

B-trees – Deletion

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

**FINAL TREE**
B-trees – Deletion

• Case 4: underflow & ‘poor sibling’ (eg., delete 13 from T0)

  ![Diagram of B-tree deletion](image)

• Merge, by pulling a key from the parent
• exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
• I.e.
B-trees – Deletion

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

A: merge w/ ‘poor’ sibling

B-trees – Deletion

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

FINAL TREE

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’
- -> ‘pull key from parent, and merge’
- Q: What if the parent underflows?
B-trees – Deletion

• Case 4: underflow & ‘poor sibling’
• \( \rightarrow \) ‘pull key from parent, and merge’
• Q: What if the parent underflows?
• A: repeat recursively

Overview

• B – trees
• B⁺-trees, B*-trees
• hashing

B⁺ trees - Motivation

if we want to store the whole record with the key \( \rightarrow \) problems (what?)
Solution: B+ - trees

- 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

```
<6     6 9
1 3 6 7

>=6
<9

>=9
<9
```

B+ trees - insertion

Eg., insert '8'

```
<6     6 9
1 3 6 7

>=6
<9

>=9
<9

1 3 6 7
```

Copyright: C. Faloutsos (2012)
Overview

- B – trees
- B+ - trees, B*-trees
- hashing

B*-trees

- splits drop util. to 50%, and maybe increase height
- How to avoid them?

B*-trees: deferred split!

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

Diagram: B*-trees structure with deferred split
B*-trees: deferred split!

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

- Notice: shorter, more packed, faster tree
- It’s a rare case, where space utilization and speed improve together
- BUT: What if the sibling has no room for our ‘lending’?

B*-trees: deferred split!

- BUT: What if the sibling has no room for our ‘lending’?
  - A: 2-to-3 split: get the keys from the sibling, pool them with ours (and a key from the parent), and 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
- B+ tree is the prevailing indexing method
- More details: [Knuth vol 3.] or [Ramakrishnan & Gehrke, 3rd ed, ch. 10]