B-Tree

CS127 TAs

** the best data structure ever
Storage Types

➔ Cache
  ◆ Fastest/most costly; volatile;

➔ Main Memory
  ◆ Fast access; too small for entire db; volatile

➔ Disk
  ◆ Long-term storage of data; random access; non-volatile

➔ Flash Memory
  ◆ No seeks (cheap reads); experimental use of dbs

➔ NVM
How Disk Works

➔ Surface divided into tracks -> sectors (smallest unit of data that can be R/W)
  ◆ Disk arm swings to position head on right track
  ◆ Platter spins continually as data is R/W from sector

➔ Measuring Disk Speed
  ◆ Access Time:
    ● Seek Time: time to find right track
    ● Latency Time: time for sector to appear under head
  ◆ Data Transfer Rate:
    ● The rate at which data can be retrieved

➔ Sequential I/O < Random I/O
Problem Statement

➔ Many queries reference only a fraction of records
➔ It is inefficient for the system to read every single tuple
➔ A better design is for the system to locate reference directly
Index

➔ Indexes
   ◆ Auxiliary data structures over relations that can improve the search time

➔ Wait, what?????
   ◆ Think about index in a textbook
   ◆ We search for an index, find corresponding pages, and then read information to find what we are looking for
   ◆ The index is much smaller than the book and has words sorted in alphabetical order

➔ Choices
   ◆ Primary v Secondary
   ◆ Dense v Sparse
   ◆ Single v Multi level
B-trees

- 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
  - Helpful to think of it as a tree structure with n-pointers in each node
B-tree nodes

- Key values are ordered
- MAXIMUM: n pointers
- MINIMUM: \([n/2]\) pointers
  - Exception: root’s minimum = 2
- Internal nodes must have \(p-1\) key values where \(p\) is the number of pointers
  - Leaf nodes must have \((n-1)/2\) key value
B-tree of order 5

- Key values appear once
- Record pointers accompany keys
Let’s run through an exact match query

Let’s find the value 16?
Let's run through an exact match query

Let's find the value 16?
Let’s run through an exact match query

Let’s find the value 16?
Let’s run through an exact match query

Let’s find the value 16?
Let’s run through a range query

Let’s find the values between 6 and 13?

Queries
Let’s run through a range query

Let’s find the values between 6 and 13?

Queries

➔ Let’s run through a range query

◆ Let’s find the values between 6 and 13?
Queries

Let’s run through a range query

- Let’s find the values between 6 and 13?
Let’s run through a range query

Let’s find the values between 6 and 13?
How To Maintain B-trees?

➔ B-tree rules must be obeyed on every insert + delete
➔ Rules:
  ◆ Insert in leaf, if room exists
  ◆ On overflow (no more room)
    ● Split: create a new internal node
    ● Redistribute keys
      ○ So that it preserves B-tree properties
      ○ Push middle key up (recursively)
Let’s run through an overflow insertion

Let’s insert the value 6:

Queries
Let’s run through an overflow insertion

- Let’s insert the value 6:
Let’s run through an overflow insertion

Let’s insert the value 6:
Let’s run through an overflow insertion

Let’s insert the value 6:
Let’s run through an overflow insertion

Let’s insert the value 6:
Overview

➔ Insert in leaf: on overflow push middle up recursively
➔ Preserves all B-tree properties
➔ Height increases when root overflows and splits
➔ Automatic, re-organization
Deletion

4 cases for deletion

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

1: delete key at leaf - no underflow

- Delete 4
Deletion

1: delete key at leaf - no underflow

- Delete 4
Deletion

➔ 2: delete key at non-leaf - no underflow

◆ Delete 15
Deletion

2: delete key at non-leaf - no underflow
- Delete 15
Deletion

3: delete key at leaf; underflow, and ‘rich sibling’

- Delete 24

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

- Underflow
- Rich sibling
Deletion

→ 3: delete key at leaf; underflow, and ‘rich sibling’

◆ Delete 24

Diagram:

- Root node: 8, 15, 21, 25
- Child nodes:
  - 8: 1, 2, 3, 4
  - 15: 9, 10, 11, 12
  - 21: 16, 18, 19
  - 25: 22, 26, 27

Arrows indicate the flow of keys during deletion and underflow handling.
Deletion

3: delete key at leaf; underflow, and ‘rich sibling’
- Delete 24
Deletion

4: delete key at leaf; underflow, and ‘poor sibling’

- Delete 26

```
1 2 3 4

9 10 11 12

16 18 19

8 15 21 25

26 27
```

>21

<25

>25

<21

>15

<15

>15

<8

>8

<8
Deletion

4: delete key at leaf; underflow, and ‘poor sibling’
- Delete 21
Deletion

4: delete key at leaf; underflow, and ‘poor sibling’

- Delete 21
B+-trees

- Allow sequential operations
  - String all leaf nodes together
  - Every key appears at leaf level (some keys show up more than once)
- This mean non-leaf nodes do not contain data for the key, only the pointer value and key value
B+-tree Insertion

Insert 5 into B-tree of order 3
B+-tree Insertion

Insert 5

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
</table>

<3 ➔ ≥3 ➔ ≥7

<7 ➔ ≥7
B+-tree Insertion

Insert 5
B+-tree Insertion

→ Insert 5

Diagram:

- Root: 3, 7
- Left subtree: 1, 2
- Middle subtree: 3, 5, 6
- Right subtree: 7, 9
B+-tree Insertion

Insert 5

Diagram of B+-tree insertion:
B+-tree Insertion

Insert 5

1 2 3 4 5 6 7 8 9

<3 ≤3 ≥3

1 2 3

<5 ≥5

5 7 9

<7 ≥7

5 6

7
B+-tree Deletion

➔ Delete 3

Diagram of B+-tree deletion process.
B+-tree Deletion

→ Delete 2
B+-tree Deletion

→ Delete 2
B+-tree Deletion

→ Delete 2
B+-tree Deletion

→ Delete 10
B+-tree Deletion

➔ Delete 10
B+-tree Deletion

→ Delete 10

```
1 9 11
10 16
16 18 19
```
B+-tree Deletion

➔ Delete 10