CPS216: Advanced Database Systems

Notes 06: Operators for Data Access (contd.)
Shivnath Babu
Insertion in a B-Tree

Insert: 62
Insertion in a B-Tree

Insert: 62
Insertion in a B-Tree

$n = 2$

Insert: 50
Insertion in a B-Tree

Insert: 50
Insertion in a B-Tree

Insert: 75
Insertion in a B-Tree

Insert: 75
Insertion
Insertion
Insertion
Insertion
Insertion
Insertion
Insertion
Insertion
Insertion
Insertion
Insertion
Insertion: Primitives

- Inserting into a leaf node
- Splitting a leaf node
- Splitting an internal node
- Splitting root node
Inserting into a Leaf Node

54 57 60 62

58
Inserting into a Leaf Node

```
54 57 60 62
```

58
Inserting into a Leaf Node

54 57 58 60 62

58
Splitting a Leaf Node
Splitting a Leaf Node

- Original node with values 54, 66
- Splitting into two new nodes
  - Left node with values 54, 57, 58, 60, 62
  - Right node with no values
Splitting a Leaf Node

61

54 66

54 57 58

60 61 62

25
Splitting a Leaf Node
Splitting a Leaf Node

61

54 59 66

54 57 58

60 61 62
Splitting an Internal Node

\[
\begin{array}{cccc}
\text{\ldots} & 21 & 99 & \text{\ldots} \\
\end{array}
\]

\[
\begin{array}{cccc}
54 & 66 & 40 & \\
\end{array}
\]

\[
\begin{array}{cccc}
74 & 84 & \\
\end{array}
\]

\[
\begin{array}{cccc}
40 & 54 & 66 & 74 & 84 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{[54, 59)} & \text{[59, 66)} & \text{[66, 74)} \\
\end{array}
\]
Splitting an Internal Node

---

[Diagram: Tree node splitting with values 40, 54, 66, 74, 84 and splitting value 59. The ranges [54, 59), [59, 66), and [66, 74) are shown.]
Splitting an Internal Node

66

[21, 66)

[54, 59)

[59, 66)

[66, 74)
Splitting the Root
Splitting the Root

\[
\begin{align*}
40 & & 54 & & 66 & & 74 & & 84 \\
\end{align*}
\]

\[
[54, 59) \quad [59, 66) \quad [66, 74)
\]
Splitting the Root

- 66
- [54, 66)
- [54, 59)
- [59, 66)
- [66, 74)
Deletion

Diagram of a deletion process in a data structure.
Deletion

redistribute
Deletion
Deletion - II
Deletion - II
Deletion - II
Deletion - II
Deletion - II
Deletion - II
Deletion - II
Deletion: Primitives

- Delete key from a leaf
- Redistribute keys between sibling leaves
- Merge a leaf into its sibling
- Redistribute keys between two sibling internal nodes
- Merge an internal node into its sibling
Merge Leaf into Sibling
Merge Leaf into Sibling

72

54 58 64

67 85

68 75
Merge Leaf into Sibling

54 58 64 68 75

... 67 85

72
Merge Leaf into Sibling

54 58 64 68 75

... 85

72
Merge Internal Node into Sibling

[Diagram showing two nodes connected by an arrow, with ranges [52, 59) and [59, 63) indicated]
Merge Internal Node into Sibling

[Diagram showing merging of internal nodes into sibling nodes with key values 41, 48, 52, 59, 63 and intervals [52, 59) and [59, 63)]
B-Tree Roadmap

- B-Tree
  - Recap
  - Insertion (recap)
  - Deletion
- Construction
  - Efficiency
- B-Tree variants
- Hash-based Indexes
Question

How does insertion-based construction perform?
B-Tree Construction

Sort

48  57  41  15  75  21  62  34  81  11  97  13
B-Tree Construction

Scan

11 13 15 → 21 34 41 → 48 57 62 → 75 81 97

11 13 15 21 34 41 48 57 62 75 81 97
B-Tree Construction

Scan
B-Tree Construction

Why is sort-based construction better than insertion-based one?
Cost of B-Tree Operations

- Height of B-Tree: $H$
- Assume no duplicates
- Question: what is the random I/O cost of:
  - Insertion:
  - Deletion:
  - Equality search:
  - Range Search:
Height of B-Tree

- Number of keys: $N$
- B-Tree parameter: $n$

$$\text{Height} \approx \log_n N = \frac{\log N}{\log n}$$

In practice: 2-3 levels
Question: How do you pick parameter n?

1. Ignore inserts and deletes
2. Optimize for equality searches
3. Assume no duplicates
Roadmap

- B-Tree
- B-Tree variants
  - Sparse Index
  - Duplicate Keys
- Hash-based Indexes
Roadmap

- B-Tree
- B-Tree variants
- Hash-based Indexes
  - Static Hash Table
  - Extensible Hash Table
  - Linear Hash Table
Hash-Based Indexes

- Adaptations of main memory hash tables
- Support equality searches
- No range searches
Indexing Problem (recap)

Index Keys

A = val

record pointers
Main Memory Hash Table

$h(key) = key \mod 8$

key

$h(key)$

buckets

0 → 32 → 48 → (null)
1 → (null)
2 → 10 → (null)
3 → 27 → 75 → (null)
4 → (null)
5 → 21 → (null)
6 → (null)
7 → 55 → (null)
Adapting to disk

- 1 Hash Bucket = 1 Block
  - All keys that hash to bucket stored in the block
  - Intuition: keys in a bucket usually accessed together
  - No need for linked lists of keys …
Adapting to Disk

How do we handle this?
Adapting to disk

- 1 Hash Bucket = 1 Block
  - All keys that hash to bucket stored in the block
  - Intuition: keys in a bucket usually accessed together
  - No need for linked lists of keys …
  - … but need linked list of blocks (overflow blocks)
Adapting to disk

- Bucket Id → Disk Address mapping
  - Contiguous blocks
  - Store mapping in main memory
    - Too large?
  - Dynamic → Linear and Extensible hash tables
Beware of claims that assume 1 I/O for hash tables and 3 I/Os for B-Tree!!