CPS216: Data-intensive Computing Systems

Operators for Data Access (contd.)

Shivnath Babu
Insertion in a B-Tree

Insert: 62
Insertion in a B-Tree

Insert: 62

$n = 2$
Insertion in a B-Tree

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: Primitives

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

| 54 | 57 | 60 | 62 |

![Diagram](image-url)
## Inserting into a Leaf Node

<table>
<thead>
<tr>
<th>54</th>
<th>57</th>
<th>58</th>
<th>60</th>
<th>62</th>
</tr>
</thead>
</table>

58
Splitting a Leaf Node

54  66

54  57  58  60  62
Splitting a Leaf Node

54  66

54  57  58  60  62

61
Splitting a Leaf Node

54 66

54 57 58

60 61 62
Splitting a Leaf Node
Splitting a Leaf Node
Splitting an Internal Node

[54, 59) [ 59, 66) [66, 74)
Splitting an Internal Node

[54, 59)  [ 59, 66)  [66,74)
Splitting an Internal Node
Splitting the Root
Splitting the Root

40  54  66  74  84

[54, 59)  [59, 66)  [66, 74)
Splitting the Root

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

redistribute
Deletion
Deletion - II
Deletion - II
Deletion - II
Deletion - II
Deletion - II
Deletion - II

Not needed

merge
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

54  58  64

...  67  85

68  75

72
Merge Leaf into Sibling

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>58</td>
<td>64</td>
<td>68</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... 67 85

54 58 64 68 75

[Diagram showing the merging process]
Merge Leaf into Sibling

\[
\begin{array}{cccccc}
54 & 58 & 64 & 68 & 75 & \ldots & 85 \\
\end{array}
\]
Merge Internal Node into Sibling

![Diagram showing the merge of internal nodes into siblings]

- [52, 59)
- [59, 63)
Merge Internal Node into Sibling

### Diagram

- **Left Node**: 41, 48, 52, 59, 63
- **Right Node**: [52, 59), [59, 63)
- **Middle Node**: ... 59 ...

- The middle node is merged into the right node, resulting in two ranges: [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
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

\[ A = \text{val} \]

\[ a_1 \rightarrow \]
\[ a_2 \rightarrow \]
\[ a_i \rightarrow \]
\[ a_n \rightarrow \]

record pointers
The image illustrates a hash table with 8 buckets. The hash function $h(key) = key \% 8$ is shown to determine the bucket for each key. The keys 32, 48, 10, 27, 75, 21, and 55 are placed in their respective buckets, and the bucket at index 3 is empty. The diagram visually represents the placement of keys in the buckets.
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
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!!