CPS216: Advanced Database Systems

Notes 05: Operators for Data Access

Shivnath Babu
Problem

- Relation: Employee (ID, Name, Dept, …)
- 10 M tuples
- (Filter) Query:

```
SELECT * 
FROM Employee
WHERE Name = "Bob"
```
Solution #1: Full Table Scan

- **Storage:**
  - Employee relation stored in *contiguous* blocks

- **Query plan:**
  - Scan the entire relation, output tuples with Name = “Bob”

- **Cost:**
  - Size of each record = 100 bytes
  - Size of relation = $10 \times 100 = 1$ GB
  - Time @ 20 MB/s $\approx 1$ Minute
Solution #2

- **Storage:**
  - Employee relation *sorted* on Name attribute

- **Query plan:**
  - Binary search
Solution #2

- **Cost:**
  - Size of a block: 1024 bytes
  - Number of records per block: $1024 / 100 = 10$
  - Total number of blocks: $10 \text{ M} / 10 = 1 \text{ M}$
  - Blocks accessed by binary search: 20
  - Total time: $20 \text{ ms} \times 20 = 400 \text{ ms}$
Solution #2: Issues

- Filters on different attributes:

  ```sql
  SELECT * 
  FROM Employee 
  WHERE Dept = "Sales"
  ```

- Inserts and Deletes
Indexes

- Data structures that efficiently evaluate a class of filter predicates over a relation
- Class of filter predicates:
  - Single or multi-attributes (*index-key attributes*)
  - Range and/or equality predicates
- (Usually) independent of physical storage of relation:
  - Multiple indexes per relation
Indexes

- Disk resident
  - Large to fit in memory
  - Persistent
- Updated when indexed relation updated
  - Relation updates costlier
  - Query cheaper
Problem

- Relation: Employee (ID, Name, Dept, …)
- (Filter) Query:

```
SELECT * 
FROM Employee 
WHERE Name = "Bob"
```

Single-Attribute Index on Name that supports equality predicates
Roadmap

- Motivation
- Single-Attribute Indexes: Overview
- Order-based Indexes
  - B-Trees
- Hash-based Indexes (May cover in future)
  - Extensible Hashing
  - Linear Hashing
- Multi-Attribute Indexes (Chapter 14 GMUW, May cover in future)
Single Attribute Index: General Construction

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$a_i$</td>
<td>$b_i$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$a_n$</td>
<td>$b_n$</td>
</tr>
</tbody>
</table>
Single Attribute Index: General Construction

A = val
A > low
A < high

A
B

<table>
<thead>
<tr>
<th>a₁</th>
<th>b₁</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>a₂</td>
<td>b₂</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>aᵢ</td>
<td>bᵢ</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>aₙ</td>
<td>bₙ</td>
</tr>
</tbody>
</table>
Exceptions

- Sparse Indexes
  - Require specific physical layout of relation
  - Example: Relation sorted on indexed attribute
  - More efficient
Single Attribute Index: General Construction

**Textbook: Dense Index**

- **A = val**
- **A > low**
- **A < high**

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1</td>
<td>b_1</td>
</tr>
<tr>
<td>a_2</td>
<td>b_2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>a_i</td>
<td>b_i</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>a_n</td>
<td>b_n</td>
</tr>
</tbody>
</table>
```
How do we organize (attribute, pointer) pairs?

Idea: Use dictionary data structures

Issue: Disk resident?
Roadmap

- Motivation
- Single-Attribute Indexes: Overview
- Order-based Indexes
  - B-Trees
- Hash-based Indexes
  - Extensible Hashing
  - Linear Hashing
- Multi-Attribute Indexes
B-Trees

- Adaptation of search tree data structure
  - 2-3 trees
- Supports range predicates (and equality)
Use Binary Search Tree Directly?

Diagram of a binary search tree with the following nodes:
- 71
- 32
- 16
- 54
- 74
- 83
- 92

The nodes are arranged such that for any node, all nodes in its left subtree have keys less than the node, and all nodes in its right subtree have keys greater than the node.
Use Binary Search Tree Directly?

- Store records of type `<key, left-ptr, right-ptr, data-ptr>`
- Remember position of root
- Question: will this work?
  - Yes
  - But we can do better!
Use Binary Search Tree Directly?

- Number of keys: 1 M
- Number of levels: $\log (2^{20}) = 20$
- Total cost index lookup: 20 random disk I/O
  - $20 \times 20 \text{ ms} = 400 \text{ ms}$

B-Tree: less than 3 random disk I/O
B-Tree vs. Binary Search Tree

1 Random I/O prunes tree by half

1 Random I/O prunes tree by 40
Meaning of Internal Node

key < 84

84 ≤ key < 91

91 ≤ key
B-Tree Example
Meaning of Leaf Nodes

- Pointer to record 63
- Pointer to record 76
- Next leaf
Equality Predicates

key = 87
Equality Predicates

key = 87
Equality Predicates

key = 87

15 36 57 63 76 87 92 100

null
Equality Predicates

key = 87

15
36
63
84
87
92
null
Range Predicates

57 ≤ key < 95

15 36 57

63 76

92 100

null
Range Predicates

57 ≤ key < 95

15 → 36 → 36 → 63 → 76 → 87 → 92 → null
Range Predicates

57 ≤ key < 95
Range Predicates

57 ≤ key < 95

15 → 36 → 63

36 → 36 → 57 → 63 → 76 → 87

84 → 91 → 92 → 100 → null
Range Predicates

57 ≤ key < 95
General B-Trees

- Fixed parameter: \( n \)
- Number of keys: \( n \)
- Number of pointers: \( n + 1 \)
B-Tree Example

n = 2
General B-Trees

- Fixed parameter: $n$
- Number of keys: $n$
- Number of pointers: $n + 1$
- All leaves at same depth
- All (key, record pointer) in leaves
B-Tree Example

n = 2

15 36 57
63 76
87
92 100
null
General B-Trees:
Space related constraints

- Use at least
  
  Root: 2 pointers
  
  Internal: $\lceil (n+1)/2 \rceil$ pointers
  
  Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data
n=3

Internal

Max

5 15 21

Min

15

Leaf

31 42 56

31 42
Leaf Nodes

- n key slots
- (n+1) pointer slots
Leaf Nodes

- **n key slots**: $k_1, k_2, k_3, \ldots, k_m$
- **(n+1) pointer slots**
- **Record of $k_1$**
- **Record of $k_2$**
- **Record of $k_m$**

Unused pointer slots:

Next leaf:

Unused:
Leaf Nodes

\[ m \geq \left\lceil \frac{n+1}{2} \right\rceil \]

- **n key slots**
- **(n+1) pointer slots**
- Record of \( k_1 \)
- Record of \( k_2 \)
- Record of \( k_m \)

Next leaf

Unused
Internal Nodes

- n key slots
- (n+1) pointer slots
Internal Nodes

- $n$ key slots
- $(n+1)$ pointer slots
- $k_1 < \text{key} < k_2$
- $k_1 \leq \text{key} < k_2$
- $k_m \leq \text{key}$

Unused
Internal Nodes

\[(m+1) \geq \left\lceil \frac{(n+1)}{2} \right\rceil\]
Root Node

\[(m+1) \geq 2\]

- **n key slots**
  - \[k_1, k_2, k_3, \ldots, k_m\]

- **(n+1) pointer slots**

- \(k_1 \leq \text{key} < k_2\)

- \(k_m \leq \text{key}\)

- **unused**
Limits

- Why the specific limits \( \lceil \frac{n+1}{2} \rceil \) and \( \lfloor \frac{n+1}{2} \rfloor \) ?
- Why different limits for leaf and internal nodes?
- Can we reduce each limit?
- Can we increase each limit?
- What are the implications?