CPS216: Data-intensive Computing Systems

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 M x 100 = 1 GB
  - Time @ 20 MB/s ≈ 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:

```
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>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$a_i$</td>
<td>$b_i$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$a_n$</td>
<td>$b_n$</td>
</tr>
</tbody>
</table>
Single Attribute Index: General Construction

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
</tr>
<tr>
<td>a_i</td>
<td>b_i</td>
</tr>
<tr>
<td>a_n</td>
<td>b_n</td>
</tr>
</tbody>
</table>

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

A = val

A > low

A < high
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

A \rightarrow B

\begin{align*}
a_1 & \rightarrow a_1 \quad b_1 \\
a_2 & \rightarrow a_2 \quad b_2 \\
\vdots & \vdots \quad \vdots \\
a_i & \rightarrow a_i \quad b_i \\
a_n & \rightarrow a_n \quad b_n
\end{align*}
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 nodes 71, 32, 54, 74, 83, 92, 16.

- Root: 71
- Left: 32, 16
  - Left: 16
- Right: 54, 83
  - Right: 54
- Right: 74, 92
  - Right: 74
- Right: 83, 92
  - Right: 92}
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
B-Tree Example
B-Tree Example
Meaning of Internal Node

- key < 84
- 84 ≤ key < 91
- 91 ≤ key
B-Tree Example

```
63
  /  
36   63
  |   /  
15 36 57 76
```

```
84   87
  |   /  
91 92 100
```
Meaning of Leaf Nodes

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

key = 87
Equality Predicates

diagram with keys 15, 36, 57, 63, 76, 87, 92, 100, and key = 87
Equality Predicates

key = 87

15  36  57
15  36  57
63
63
null
null
84  91
84  91
null
null
87
87
null
null
92  100
92  100
null
null
Equality Predicates

key = 87

15

36

63

84 91

87

100

null
Range Predicates

57 ≤ key < 95

15
36
63
null

36
57

84
91

63
76

87

92
100
Range Predicates

57 ≤ key < 95

15
36 57
63
84 91
63 76
87
92 100
null
Range Predicates

$57 \leq \text{key} < 95$
Range Predicates

57 ≤ key < 95
Range Predicates

57 ≤ key < 95
Range Predicates

57 ≤ key < 95

15

36

36 57

63

63 76

84 91

87

92 100

null
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
General B-Trees: Space related constraints

- Use at least

  Root: 2 pointers

  Internal: \( \lceil \frac{n+1}{2} \rceil \) pointers

  Leaf: \( \lfloor \frac{n+1}{2} \rfloor \) pointers to data
n=3

Max

5 15 21

Internal

Min

15

Leaf

31 42 56

31 42
Leaf Nodes

n key slots

(n+1) pointer slots
Leaf Nodes

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

<table>
<thead>
<tr>
<th>k_1</th>
<th>k_2</th>
<th>k_3</th>
<th>...</th>
<th>...</th>
<th>k_m</th>
</tr>
</thead>
</table>

unused

record of k_1
record of k_2
record of k_m

next leaf
Leaf Nodes

\[ m \geq \left\lfloor \frac{(n+1)}{2} \right\rfloor \]

- \( n \) key slots
- \((n+1)\) pointer slots
- unused

\begin{array}{cccccc}
  k_1 & k_2 & k_3 & \ldots & \ldots & k_m \\
\end{array}

record of \( k_1 \)
record of \( k_2 \)
record of \( k_m \)

next leaf
Internal Nodes

n key slots

(n+1) pointer slots
Internal Nodes

- **n** key slots: \(k_1, k_2, k_3, \ldots, k_m\)
- **(n+1)** pointer slots
- Key values:
  - \(k_1 \leq \text{key} < k_2\)
  - \(k_1 \leq \text{key} < k_2\)

Unused slots indicate missing keys or placeholders.
Internal Nodes

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

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

unused
Root Node

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

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

key < \(k_1\)

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

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

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