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 M / 10 = 1 M
- Blocks accessed by binary search: 20
- Total time: 20 ms x 20 = 400 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></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td></td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td></td>
<td>$b_2$</td>
</tr>
<tr>
<td></td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td>$a_i$</td>
<td></td>
<td>$b_i$</td>
</tr>
<tr>
<td></td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td>$a_n$</td>
<td></td>
<td>$b_n$</td>
</tr>
</tbody>
</table>
Single Attribute Index: General Construction

A = val
A > low
A < high

A

\[ A = \text{val} \]
\[ A > \text{low} \]
\[ A < \text{high} \]
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

<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_n)</td>
<td>(b_n)</td>
</tr>
</tbody>
</table>

- **A = val**
- **A > low**
- **A < high**
Single Attribute Index: General Construction

How do we organize \((\text{attribute, pointer})\) pairs?

Idea: Use dictionary data structures

Issue: Disk resident?

\begin{align*}
A &= \text{val} \\
A &> \text{low} \\
A &< \text{high}
\end{align*}
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?

16, 32, 54, 71, 74, 83, 92

16, 32, 54, 71, 74, 83, 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:** $1 \text{ M}$
- **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

15 36 57 63 76 87 92 100
B-Tree Example
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

The diagram illustrates a tree structure with keys 15, 36, 57, 63, 76, 87, 92, and 100. The key value 87 is highlighted and is found in the tree.
Equality Predicates

key = 87
Equality Predicates

key = 87
Equality Predicates

key = 87
Range Predicates

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

57 ≤ key < 95

Diagram:

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

\[ 57 \leq key < 95 \]
Range Predicates

57 ≤ key < 95
Range Predicates

57 ≤ key < 95
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

\[ n = 2 \]

```
63
/  \
36   63
/  \
15   36 57
|   /  \   |
|  /     \  |
| /       \ |
15       36 57
|   /  \   |
|  /     \  |
| /       \ |
null
```

```
General B-Trees:
Space related constraints

- Use at least
  
  **Root:** 2 pointers
  
  **Internal:** $\lceil (n+1)/2 \rceil$ pointers
  
  **Leaf:** $\lceil (n+1)/2 \rceil$ 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
- (n+1) pointer slots
  - record of \(k_1\)
  - record of \(k_2\)
  - record of \(k_m\)

Unused

next leaf
Leaf Nodes

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

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

Unused

Next leaf
**Internal Nodes**

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

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

<table>
<thead>
<tr>
<th>k₁</th>
<th>k₂</th>
<th>k₃</th>
<th>...</th>
<th>kₘ</th>
</tr>
</thead>
</table>

- **key < k₁**
- **k₁ ≤ key < k₂**
- **kₘ ≤ key**
- **unused**
Internal Nodes

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

Unused

\(n\) key slots

\((n+1)\) pointer slots

key < \(k_1\)

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

\(k_m \leq \text{key}\)
Root Node

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

- n key slots
- (n+1) pointer slots
- key < k₁
- \(k₁ \leq \text{key} < k₂\)
- \(kₘ \leq \text{key}\)

Unused
Limits

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