Topics

- Indexing
- B+Trees
- Extendible Hash Index
- Bitmap Index
- Conclusion
Bitmap Indexing (David's DB Research)

- **Bitmaps** are commonly used to index high-dimensional data
  - Idea: Discretize search-key attributes into a set of "bins"
  - (*e.g.*, Oracle, Apache Hive, FastBit, ...)

- Advantages:
  - Exploit CPU's fast bit-wise operations to resolve queries
  - Highly compressible
  - Paradox: query speedup proportional to compression ratio (*How?*)
## Bitmap Example

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Name</th>
<th>Age</th>
<th>Salary</th>
<th>City</th>
<th>Favorite Drink</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Sara</td>
<td>42</td>
<td>65,000</td>
<td>Beaverton</td>
<td>Ninkasi</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>Julie</td>
<td>50</td>
<td>130,000</td>
<td>West Linn</td>
<td>Bridgeport</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>Tom</td>
<td>21</td>
<td>25,000</td>
<td>Portland</td>
<td>Boneyard</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Binning

<table>
<thead>
<tr>
<th>Name</th>
<th>Age &lt;= 25</th>
<th>25 &lt; Age &lt; 50</th>
<th>50 &lt;= Age</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sara</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Julie</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Tom</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Query processing is *fast (sometimes)*

- Find everyone named “Julie” who’s under 40 years of age

<table>
<thead>
<tr>
<th>Name</th>
<th>Age &lt; 25</th>
<th>25 &lt; Age &lt; 50</th>
<th>50 &lt;= Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sara?</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Julie?</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Tom?</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sara?</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Julie?</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Tom?</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$$
\begin{bmatrix}
0 \\
1 \\
0 \\
\ldots
\end{bmatrix} \&
\begin{bmatrix}
0 \\
0 \\
1 \\
\ldots
\end{bmatrix} =
\begin{bmatrix}
0 \\
1 \\
0 \\
\ldots
\end{bmatrix}
$$

(prune)

To Disk

(prune)
Querying is *fast (sometimes)*

- Find everyone making between $24,000 and $50,000 salary

```
  | Salary |
  |< 25,000| >=25,000|
  | 0      | 1       |
  | 0      | 1       |
  | 1      | 0       |
  | ...    |         |
```

To Disk

To Disk

To Disk
Attribute-binning matters

- More ranges (more bit-vectors)
  - Pros: Approximates exact-match
  - Cons: Diminishing returns and size increases
    - Vector per name in previous example seems like overkill
  - Have to get the bin-ranges right (could add bit-vectors that don't add selectivity)

- Coarser ranges (fewer bit-vectors)
  - Pros: Size
  - Cons: Approximates file scan
    - Worse, random disk access depending on tuple distribution in files
Bitmap Takeaways

- Bitmap size: \( O(mn) \)
  - Troubling for high-dimensional (e.g., many attributes) data sets:
    - \( n = \) Attributes \(*\) discretization factor per attribute
    - \( m = \) tuples (observations, transactions, etc.) -- large!
      - To compare two vectors, we read \( 2 \times \frac{m}{w} \) words from cache or memory
      - \( w = \) word size
  - But comparison is fast: bitwise operation processes \( w \)-bits at a time
Bitmap Takeaways

- Opposing goals:
  - Minimize disk scans (finer-grained bins)
  - Still want bitmap index to fit in cache or core memory (coarser bins)

- But bit vectors are long and sparse... what if we compress each vector?
  - Typically a bad idea (decoding overhead)
  - *(But if we didn’t have to decode...)*

<table>
<thead>
<tr>
<th>50 &lt;= Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Topics

- Indexing
- B+Trees
- Extendible Hashing
- Bitmaps
  - Bitmap Compression
- Conclusion
Word-Aligned Hybrid Code (WAH)

- Allow \textit{both} run-length and literal bit-strings to be encoded
  - Code size is fixed to CPU-word length (32-bits in our examples)

- Allows for two types of code formats:
  - Literal Word: \[\text{0x}\ldots\text{x}\]
    - Encodes 31 bits of literal string: \textit{xxxxxx}...\textit{x}
    - \text{01010101...10} is really: "1010101...10"
  - Fill Word: \[\text{1F}\ldots\text{x}\]
    - Encodes a run of \(31 \times (\text{xxxxxx}...\text{x})\) \textit{F}-bits
    - \text{10000...01111} is really: 31*15 (= 465) consecutive 0s
Example: WAH Compression

- **raw bit-vector**
  - 1
  - 0
  - 1
  - 0
  - 0
  - ... (all 0s)
  - 1

- **Organize in groups of 31-bits**
  - 31-bits

- **Merge neighboring groups into group-runs**
  - 31-bits of literal
  - 58 groups of 31-zeros
  - 31-bits of literal

- **Encode as WAH word**
  - 010100000..1
  - 1000..111010
  - 011111111..1
Consider taking a logical `&' over the following two vectors: v1 & v2

- 1860 bits (rows) are represented by each vector

<table>
<thead>
<tr>
<th></th>
<th>lit (31 bits)</th>
<th>1798 zeros</th>
<th>lit (31 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>010100..0</td>
<td>100..111010</td>
<td>0111111..1</td>
</tr>
<tr>
<td>v2</td>
<td>00010..01</td>
<td>001..110110</td>
<td>110..0111001</td>
</tr>
</tbody>
</table>

- 399 zeros
WAH Query Processing Example

- Step 1: Fetch first word from both vectors.
  - These two are now known as "active words"

\[ \begin{align*}
    \text{v1} & \begin{array}{c}
        010100..0 \\
        100..111010 \\
        0111111..1
    \end{array} \\
    \text{v2} & \begin{array}{c}
        00010..01 \\
        001..110110 \\
        110..0111001 \\
        000..010100
    \end{array}
\end{align*} \]

result
WAH Query Processing Example

- Step 2: Decode them. They’re both literal words!
  - So apply the `&' directly between the words

\[
\begin{align*}
\text{v1} & : 010100..0 \quad 100..111010 \quad 0111111..1 \\
\text{v2} & : 00010..01 \quad 001..110110 \quad 110..0111001 \quad 000..010100 \\
\text{result} & : 000100..0
\end{align*}
\]
WAH Query Processing Example

- Step 3: Fetch next two words
- Step 4: Decode: one fill-word and one literal word!

\[ \begin{align*}
\text{v1} &\quad 010100\ldots0 & 100\ldots111010 & 0111111\ldots1 \\
\text{v2} &\quad 00010\ldots01 & 001\ldots110110 & 110\ldots0111001 & 000\ldots010100 \\
\end{align*} \]

A run of 58*31 (= 1798) consecutive zeros

31 bit literal

result \[ 000100\ldots0 \]
WAH Query Processing Example

- Step 4a: Apply bit-wise & with the fill-bit (0)

(This step demonstrates importance of 31-bit groupings)

A run of $58 \times 31 = 1798$ consecutive zeros

\[
\begin{align*}
v1 & \quad 010100..0 & 100..111010 & 0111111..1 \\
v2 & \quad 00010..01 & 001..110110 & 110..0111001 \quad 000..010100 \\
\text{result} & \quad 000100..0 & 000000..0
\end{align*}
\]
WAH Query Processing Example

- Step 5: v1 fill-word is still *active*, so only fetch next word from v2

```
v1  010100..0  100..111010  0111111..1
v2  00010..01  001..110110  110..0111001  000..010100
```

We've expended 31 zeroes from this fill word! Now 57*31 (= 1767) consecutive zeros

```
result  000100..0  000000..0  10..0111001
```
WAH Query Processing Example

After step 5: v1's and v2's fill-words no longer *active*, so fetch next words from both

- They're both literals, so apply the & directly again

\[
\begin{align*}
\text{v1} & : 010100..0 \quad 100..111010 \quad 0111111..1 \\
\text{v2} & : 00010..01 \quad 001..110110 \quad 110..0111001 \quad 000..010100
\end{align*}
\]

\[
\begin{align*}
\text{result} & : 000100..0 \quad 000000..0 \quad 10..0111001 \quad 000..010100
\end{align*}
\]
Topics

- Indexing
- B+Trees
- Extendible Hashing
- Bitmaps
- Conclusion
In Conclusion...

- Ordered Index vs. Hash Index: Which Is Better?
  - Depends on workload characteristics!
    - Cost of periodic index/file reorganization
    - Frequency of insertions and deletes
    - What are the dominant "SELECT" query types?
      - Lots of range queries: Use ordered indices (B+-Tree)
      - Lots of equality queries: Use hash index
      - Or, use both!

- Most databases implement both. Choice left up to DBA.
  - Pro tip: Be an informed DBA
In Conclusion... (Cont.)

- Are there other index structures? You bet!
  - R-Trees (For geospatial 2D data)
  - KD-Trees (For multi-dimensional data)
  - Grid Files (For multi-dimensional data)
  - Linear Hashing (Extendible hashing's cousin)
  - Consistent Hashing (used in P2P networks, like BitTorrent)

- So many more... (large area of research, even still)
Reminders:

- Hwk 6 (Joins!) due tonight
- Offline team meetings this week
- Capstone party tomorrow (4-6pm) in this room

Last time.. B+Tree multi-level index

- Properties
- Excels for
  - Exact match (point) queries
  - Range queries