Bitmap Indexes

Ben McCamish
Problems

• Ordered Indexes provide good update speeds but...
  ‣ They consume a lot of space
  ‣ Require the database to be sorted

• B+Trees provide fast lookup but...
  ‣ Still require a lot of space
  ‣ Queries can often be slower than other indexes
What Bitmaps Offer

• Much smaller size of index, due to compression

• Fast queries, due to bit operations and size

• They are restricted though…
  ‣ Not made for frequent updates
  ‣ High cardinality of the attributes can be a problem
Why use bitmaps?

• Static Databases
  ‣ Lots of databases are static, especially scientific ones
  ‣ BPA data is huge

• **Space** is cheap, **time** is not
  ‣ Searching large databases can be time consuming for users
What are Bitmap Indexes?

- Bitmaps are often used to represent images
- Although, a bitmap is just an array/matrix of bits
- In this class, when I say Bitmap what I actually mean is Bitmap Index
Bitmap Creation

• Bitmaps are created by first observing the possible values that an attribute can take.

• Ex. Attribute *Age* may have the range [0,120].

• The possible number of values is called *Cardinality*.
  - *Distance* might be an attribute with *high* cardinality.
  - *Graduated* might be an attribute with *low* cardinality.
## Bitmap Creation

<table>
<thead>
<tr>
<th>Adopted</th>
<th>Species</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Dog</td>
<td>12</td>
</tr>
<tr>
<td>Yes</td>
<td>Turtle</td>
<td>20</td>
</tr>
<tr>
<td>Yes</td>
<td>Cat</td>
<td>1</td>
</tr>
<tr>
<td>No</td>
<td>Cat</td>
<td>4</td>
</tr>
</tbody>
</table>
### Bitmap Creation

<table>
<thead>
<tr>
<th>Adopted</th>
<th>Species</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Dog</td>
<td>12</td>
</tr>
<tr>
<td>Yes</td>
<td>Turtle</td>
<td>20</td>
</tr>
<tr>
<td>Yes</td>
<td>Cat</td>
<td>1</td>
</tr>
<tr>
<td>No</td>
<td>Cat</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
<th>Dog</th>
<th>Turtle</th>
<th>Cat</th>
<th>0-10</th>
<th>11-20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Single 1 per tuple per attribute, rest 0s
- Index may even be larger than database, *initially*
Bitmap Creation

- Adopted attribute is binary, so very low cardinality
Bitmap Creation

- Assuming we only have three species then Species is low cardinality
- What if we don’t know how many species?
### Bitmap Creation

<table>
<thead>
<tr>
<th>Adopted</th>
<th>Species</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Dog</td>
<td>12</td>
</tr>
<tr>
<td>Yes</td>
<td>Turtle</td>
<td>20</td>
</tr>
<tr>
<td>Yes</td>
<td>Cat</td>
<td>1</td>
</tr>
<tr>
<td>No</td>
<td>Cat</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adopted</th>
<th>Species</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Dog</td>
<td>12</td>
</tr>
<tr>
<td>Yes</td>
<td>Turtle</td>
<td>20</td>
</tr>
<tr>
<td>Yes</td>
<td>Cat</td>
<td>1</td>
</tr>
<tr>
<td>No</td>
<td>Cat</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
<th>Dog</th>
<th>Turtle</th>
<th>Cat</th>
<th>0-10</th>
<th>11-20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Age is a tricky one, since the range could be large
- Even with max age of 20, lots to consider
  - Do we consider all real numbers?
Bitmap Creation

- This is where binning comes in
- Age is ‘binned’ into two bins
  - Could be more or less
Bitmap Creation

<table>
<thead>
<tr>
<th>Adopted</th>
<th>Species</th>
<th>Age</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Dog</td>
<td>12</td>
<td>Fin</td>
</tr>
<tr>
<td>Yes</td>
<td>Turtle</td>
<td>20</td>
<td>Dakota</td>
</tr>
<tr>
<td>Yes</td>
<td>Cat</td>
<td>1</td>
<td>Sky</td>
</tr>
<tr>
<td>No</td>
<td>Cat</td>
<td>4</td>
<td>Zoe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
<th>Dog</th>
<th>Turtle</th>
<th>Cat</th>
<th>0-10</th>
<th>11-20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

• What if we added names?

• How do we make a bitmap of this?
## Bitmap Creation

### Adopted Species Age Name

<table>
<thead>
<tr>
<th>Adopted</th>
<th>Species</th>
<th>Age</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Dog</td>
<td>12</td>
<td>Fin</td>
</tr>
<tr>
<td>Yes</td>
<td>Turtle</td>
<td>20</td>
<td>Dakota</td>
</tr>
<tr>
<td>Yes</td>
<td>Cat</td>
<td>1</td>
<td>Sky</td>
</tr>
<tr>
<td>No</td>
<td>Cat</td>
<td>4</td>
<td>Zoe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
<th>Dog</th>
<th>Turtle</th>
<th>Cat</th>
<th>0-10</th>
<th>11-20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- We can’t really…
  - There is no finite number of names,
  - Finite number of species, even though it’s large
  - Can create with just the names seen, not ideal though
There are drawbacks

• Bitmaps excel at databases with low cardinality
  ‣ High cardinality is possible, but some decisions need to be made on either binning or restrictions

• Impose a finite range for attributes with high cardinality or not index over them

• What if we don’t index over an attribute?
Compression

• Remember how the bitmap was larger than data?
  ‣ Not very useful when you want to store the index in memory

• Compression fixes this
  ‣ Uses Run Length Encoding (RLE)
  ‣ WAH, PLWAH, VAL, BBC, lots more…
Compression

• Run length encoding is very useful in bitmaps
  ‣ Encodes runs of repeated values as counts of how many
  ‣ Many different ways to count the runs of certain runs

• Bitmaps have many repeated values
  ‣ Since there is only a single 1 for each attribute, lots of 0s
Compression
Common Appearances

• **Words**
  - How big each encoding is, most defined by hardware

• **Runs** are sequences of either 1s or 0s greater than or equal to word size
  - Ex. Word size 8, 00000000 is a run, 00000001 is not

• **Dirty bits** can ruin compression
  - Words containing dirty bits or a mixture often called *literals*

• Columns are usually compressed, why?
One of the first

- Byte-aligned Bitmap Compression (BBC)
- Developed in 1995, actually a patent out for it
- Very good compression rate
- Complicated rules though…
Byte-aligned Bitmap Compression

• Word size is a Byte (hence the name)

• Counts 0s that appear, a byte filled with 0s is run

• A Chunk is encoded gap bytes followed by literals

• Gaps, Literals, Chunks
Byte-aligned Bitmap Compression

• Header Byte:
  ‣ Bits 1-3: Number of runs (the 0-6 size)
  ‣ Bit 4: Indicates whether the tail is special
  ‣ Bits 5-8: Number of literals at end of chunk or dirty bit location
Byte-aligned Bitmap Compression

• Run Byte(s)
  ‣ If run length is 7-127 bytes or 128-32767 bytes
  ‣ Only occurs if first three bits in header are 111 (filled up)

• Literals
  ‣ Stored exactly as they appear
  ‣ Only occurs when byte is mixture
  ‣ Number is indicated in header byte (bit 4=0)
Byte-aligned Bitmap Compression

- Three lengths of runs
  - 0-6: Stored in header byte
  - 7-127: One byte follows header byte counting gaps
  - 128-32767: Two bytes follows header byte counting gaps
Byte-aligned Bitmap Compression

• Special Bytes:
  ‣ When there is only a single byte that is a literal in the chunk
  ‣ Literal only has a dirty bit
  ‣ Stored as which bit is set

• Otherwise stored as a literal, requires another Byte
Byte-aligned Bitmap Compression Example

- Lots to process, so let's look at an example
Byte-aligned Bitmap Compression Example

- Two runs, two bytes
  - Remember word size is 8
Byte-aligned Bitmap
Compression Example

- Uhoh… dirty bit
  - Following bytes are runs, so looks like we have our first chunk!
Byte-aligned Bitmap Compression Example

- Header
  - Bits 1-3: Number of run bytes 010
Byte-aligned Bitmap Compression Example

- **Header**
  - **Bit 4**: Is it special?
  - Yes, since there is a dirty bit, so 1
Byte-aligned Bitmap Compression Example

- Header
  - Bits 5-8: Since bit 4 is set, where is the dirty bit?
  - 4th position, so 0100
Byte-aligned Bitmap Compression Example

- First chunk
  - Number of runs, Special literal, Dirty bit location

<table>
<thead>
<tr>
<th>00000000 00000000 00100000 00...00 10101010 11110000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed Version So Far</td>
</tr>
<tr>
<td>01010100</td>
</tr>
</tbody>
</table>
Byte-aligned Bitmap Compression Example

- Now we have 13 run bytes
Byte-aligned Bitmap Compression Example

- Followed by a literal
- Next sequence is also a literal
  - Another chunk!
Byte-aligned Bitmap Compression Example

- Header
  - Bits 1-3: 13 run bytes, so more than will fit
  - Set all 3, 111
Byte-aligned Bitmap Compression Example

- Header
  - Bit 4: Is the literal special?
    - No, so we set to 0
Byte-aligned Bitmap Compression Example

- **Header**
  - **Bits 5-8**: Since 4 is not set, how many literal bytes?
  - 2 literal bytes, so **0010**
Byte-aligned Bitmap Compression Example

- Run Bytes
  - Since we have more than 6 run bytes, need to count them
  - 13 run bytes, so only 1 run byte encoding, 00001101
Byte-aligned Bitmap Compression Example

- Literal Bytes
  - We have 2 literals as indicated by the header
  - Encode exactly as they appear, 10101010 11110000
Byte-aligned Bitmap Compression Example

- Chunk 1
  - Encoded 24 bits in 8
Byte-aligned Bitmap Compression Example

- Chunk 2
  - Encoded 120 bits in 32 bits
Joke Time!

• A programmer is told to “Go to hell!” They find the worst part about that statement is the “Go to”
Moving on

• Word Aligned Hybrid (WAH)

• Takes up 50% more space compared to BBC

• Performs queries 12 times faster on practical and synthetic databases!

• Much much simpler than BBC
Word Aligned Hybrid

- Uses word size based on hardware
  - 32, 64, someday maybe even 128 bits!
  - Can define any word size though
  - Why is using word size based on hardware a good idea?
- If word size is 32, then encodes runs/literals of 31
Word Aligned Hybrid

• Header is actually one 1 or two bits

• Run
  - $1000000000000010 = 0000000000000000 0000000000000000$
  - $1100000000000010 = 1111111111111111 1111111111111111$

• Literal
  - $00000110000000010 = 0000110000000010$
• Same bit string as before, only this time we will use 32 bit words
Word Aligned Hybrid Example

- Look at 31 bit sections and process them
Word Aligned Hybrid Example

\[00000000\ 00000000\ 00010000\ 00...00\ 10101010\ 11110000\]

Compressed Version of Selected

\[00000000\ 00000000\ 00010000\ 00000000\]

- First 31 bits is a mix, so store as a literal
Word Aligned Hybrid Example

• Since we took some out of here, only 97 bits now
Word Aligned Hybrid Example

97 - 31 = 66 bits left

Now onto the next 31 bits

All 0s so we start a run
Word Aligned Hybrid Example

66 - 31 = 35 bits left

| 00000000 00000000 00010000 00…00 10101010 11110000 |

Compressed Version of Selected

| 10000000 00000000 00000000 00000010 |

- Now onto the next 31 bits
- Still only 0s, so we increment the count of our run
Word Aligned Hybrid Example

35 - 31 = 4 bits left

00000000 00000000 00010000 00…00 10101010 11110000

Compressed Version of Selected

10000000 00000000 00000000 00000011

- Now onto the next 31 bits
- Still only 0s, so we increment the count of our run, again
### Word Aligned Hybrid Example

35 - 31 = 4 bits left

| 00000000 00000000 00010000 | 00...00 10101010 11110000 |

Compressed Version of Selected

- There are only 12 bits left
  - How do we store these now?
Word Aligned Hybrid Example

35 - 31 = 4 bits left

Since all the columns have same number of tuples

- We can just encode it as a shorter literal
Variable Aligned Length

- **Variable Aligned Length (VAL)**
- Fixes some of the shortcomings of WAH
  - WAH is susceptible to dirty bits, why?
- I worked on this one a little during my M.S.
- Very similar to WAH, so we will be brief
Variable Aligned Length

- VAL introduces Segments into words
  - Basically miniature words inside words
- 16 bit word, 7 bit segments
  - 01 1010001 1000010 = 1010001 1111111 1111111
Variable Aligned Length

• VAL introduces Segments into words
  ‣ Basically miniature words inside words

• 16 bit word, 7 bit segments
  ‣ $0110100011000010 = 1010001111111111111111$
Variable Aligned Length

• VAL introduces Segments into words
  ‣ Basically miniature words inside words

• 16 bit word, 7 bit segments
  ‣ 01 1010001 1000010 = 1010001 1111111 1111111
Variable Aligned Length

• VAL introduces Segments into words
  ‣ Basically miniature words inside words

• 16 bit word, 7 bit segments
  ‣ 01 1010001 1000010 = 1010001 1111111 1111111

• 32 bit word, 7 bit segments, 4 segments

• 32 bit word, 15 bit segments, 2 segments
Variable Aligned Length

• VAL tries to improve on WAHs compression

• Same query times as WAH in most cases

• Smaller segment sizes mean better compression
  ‣ To a point...
  ‣ Why do you think that is?
Queries

• Bitmap Querying happens between columns
  ‣ Really fast since all the operations are bit operations
  ‣ Select columns that you want to compare

• Range and Point queries
  ‣ AND handles specific
  ‣ OR handles range
Queries

- Bit operations between literals are simple

- Bit operations between compressed runs are still simple
  - Take the minimum run size
  - Then perform the bit operation on the two fill numbers
Queries

Want to find all Dogs and Cats
1860 bits/tuples

Dog: 01010101..01 1000..0111010 011111111..11

Cat: 01111111..11 010101010..01 1000..0111001 011111111..11
Queries

Dog & Cat

Dog: 01010101..01
&
Cat: 01111111..11

Result: 010101010..01
Queries

Dog: 01010101..01
     1000..0111010
     01111111..11

Cat: 01111111..11
     010101010..01
     1000..0111001
     011111111..11

Result: 010101010..01
        000000000..00

26 runs = 806 bits

Dog & Cat
Queries

Dog: 01010101..01

1000..0111001

011111111..11

Cat: 01111111..11

010101010..01

1000..0111001

011111111..11

25 runs = 775 bits

Dog & Cat

Result: 010101010..01

0000000000..00

1000..0111001

011111111..11
Queries

Dog & Cat

Dog: 01010101..01 1000..0111001 011111111..11

Cat: 01111111..11 010101010..01 1000..0111001 011111111..11

Result: 010101010..01 000000000..00 1000..0111001 011111111..11
## Queries

**Dog & Cat**

<table>
<thead>
<tr>
<th>Dog</th>
<th>01010101..01</th>
<th>1000..0111001</th>
<th>011111111..11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>011111111..11</td>
<td>010101010..01</td>
<td>1000..0111001</td>
</tr>
</tbody>
</table>

**Result:**

| 010101010..01 | 000000000..00 | 1000..0111001 | 011111111..11 |
Sorting

• Sorting bitmap indexes can drastically improve compression ratios

• Grey Code sorting can be performed on a bitmap

• Lexicographical sorting on the data itself is often just as effective

• But wait… didn’t we say that one of the pros of Bitmap Indexing was that sorting wasn’t required?