Histogram-Aware Sorting for Enhanced Word-Aligned Compression in Bitmap Indexes

Owen Kaser\textsuperscript{1}, Daniel Lemire\textsuperscript{2}, Kamel Aouiche\textsuperscript{2}

1- University of New Brunswick, Saint John
2- Université du Québec at Montréal (UQAM)

October 23, 2008
SELECT * FROM T WHERE x=a AND y=b;

Above, compute
\[ \{ r \mid r \text{ is the row id of a row where } x = a \} \cap \{ r \mid r \text{ is the row id of a row where } y = b \} \]
SELECT * FROM T WHERE x=a AND y=b;

Bitmap indexes have a long history. (1972 at IBM.)

Above, compute

\[
\{ r \mid r \text{ is the row id of a row where } x = a \} \cap \\
\{ r \mid r \text{ is the row id of a row where } y = b \}
\]
SELECT * FROM T WHERE x=a AND y=b;

- Bitmap indexes have a long history. (1972 at IBM.)
- Long history with DW & OLAP. (Sybase IQ since mid 1990s).

Above, compute

\[ \{ r \mid r \text{ is the row id of a row where } x = a \} \cap \{ r \mid r \text{ is the row id of a row where } y = b \} \]
• Computing the union of two sets of integers between 1 and 64 (eg row ids, trivial table)...
• E.g., \{1, 5, 8\} \cup \{1, 3, 5\}?
• Computing the union of two sets of integers between 1 and 64 (eg row ids, trivial table)...
• E.g., \{1, 5, 8\} \cup \{1, 3, 5\}? 
• Can be done in **one operation** by a CPU: 
  BitwiseOR( 10001001, 10101000)
- Computing the union of two sets of integers between 1 and 64 (eg row ids, trivial table)...
- E.g., \{1, 5, 8\} \cup \{1, 3, 5\}?  
- Can be done in **one operation** by a CPU: BitwiseOR( 10001001, 10101000) 
- Extend to sets from 1..N using \( \lceil N/64 \rceil \) operations.
A column with $n$ rows and $L$ distinct values $\Rightarrow nL$ bits

柱状图示例：

- $n$: 行数
- $L$: 值的个数
- $X$: 柱状图
Bitmap compression

- A column with $n$ rows and $L$ distinct values $\Rightarrow nL$ bits
- E.g., $n = 10^6$, $L = 10^4 \rightarrow 10$ Gbits
Bitmap compression

- A column with $n$ rows and $L$ distinct values $\Rightarrow nL$ bits
- E.g., $n = 10^6$, $L = 10^4 \rightarrow 10$ Gbits
- Uncompressed bitmaps are often impractical
A column with $n$ rows and $L$ distinct values $\Rightarrow nL$ bits

- E.g., $n = 10^6$, $L = 10^4 \rightarrow 10$ Gbits

- Uncompressed bitmaps are often impractical

- Moreover, bitmaps often contain long streams of zeroes...
A column with $n$ rows and $L$ distinct values $\Rightarrow nL$ bits

E.g., $n = 10^6$, $L = 10^4 \rightarrow 10$ Gbits

Uncompressed bitmaps are often impractical

Moreover, bitmaps often contain long streams of zeroes...

Logical operations over these zeroes is a waste of CPU cycles.
How to compress bitmaps?

- Must handle **long streams of zeroes** efficiently ⇒ Run-length encoding? (RLE)
How to compress bitmaps?

- Must handle long streams of zeroes efficiently $\Rightarrow$ Run-length encoding? (RLE)
- RLE variants can focus on runs that align with machine-word boundaries.
How to compress bitmaps?

- Must handle **long streams of zeroes** efficiently ⇒ Run-length encoding? (**RLE**)
- RLE variants can focus on runs that align with machine-word boundaries.
- Trade compression for speed.
How to compress bitmaps?

- Must handle **long streams of zeroes** efficiently ⇒ Run-length encoding? (**RLE**)  
- RLE variants can focus on runs that align with machine-word boundaries.  
- Trade compression for speed.  
- Our **EWAH** extends Wu et al.’s word-aligned hybrid.
How to compress bitmaps?

- Must handle **long streams of zeroes** efficiently \(\Rightarrow\) Run-length encoding? (RLE)
- RLE variants can focus on runs that align with machine-word boundaries.
- Trade compression for speed.
- Our **EWAH** extends Wu et al.’s word-aligned hybrid.

\[
\begin{array}{cccc}
0101000000000000 & 000...000 & 000...000 & 0011111111111100 \\
\hline
\end{array}
\]

\(\Rightarrow\) dirty word, run of 2 “clean 0” words, dirty word...
Computational and storage bounds

- $n \rightarrow$ number of rows, $c \rightarrow$ number of 1s per row;
Computational and storage bounds

- \( n \rightarrow \) number of rows, \( c \rightarrow \) number of 1s per row;
- Construction in time \( O(nc) \);

Bounds do not depend on the number of bitmaps. Implementation scales to millions of bitmaps.
Computational and storage bounds

- \( n \rightarrow \) number of rows, \( c \rightarrow \) number of 1s per row;
- Construction in time \( O(nc) \);
- Total size of bitmaps is also in \( O(nc) \);
Computational and storage bounds

- $n \rightarrow$ number of rows, $c \rightarrow$ number of 1s per row;
- Construction in time $O(nc)$;
- Total size of bitmaps is also in $O(nc)$;
- Given two bitmaps $B_1, B_2$ of compressed size $|B_1|$ and $|B_2|$. . .
Computational and storage bounds

- $n \rightarrow$ number of rows, $c \rightarrow$ number of 1s per row;
- Construction in time $O(nc)$;
- Total size of bitmaps is also in $O(nc)$;
- Given two bitmaps $B_1, B_2$ of compressed size $|B_1|$ and $|B_2|$...
- AND, OR, XOR in time $O(|B_1| + |B_2|)$. 

Bounds do not depend on the number of bitmaps.
Implementation scales to millions of bitmaps.
Computational and storage bounds

- $n \rightarrow$ number of rows, $c \rightarrow$ number of 1s per row;
- Construction in time $O(nc)$;
- Total size of bitmaps is also in $O(nc)$;
- Given two bitmaps $B_1, B_2$ of compressed size $|B_1|$ and $|B_2|$…
- AND, OR, XOR in time $O(|B_1| + |B_2|)$.
- **Bounds do not depend on the number of bitmaps.** Implementation scales to *millions* of bitmaps.
RLE, BBC, WAH, EWAH are order-sensitive: they compress sorted data better;
Improving compression by sorting the table

- **RLE, BBC, WAH, EWAH** are order-sensitive: they compress **sorted** data better;
- But finding the *best* row ordering is NP-hard.
Improving compression by sorting the table

- RLE, BBC, WAH, EWAH are order-sensitive: they compress sorted data better;
- But finding the best row ordering is NP-hard.
- Lexicographic sorting is
  - fast, even for very large tables.
Improving compression by sorting the table

- **RLE, BBC, WAH, EWHAH** are order-sensitive: they compress *sorted* data better;
- But finding the *best* row ordering is NP-hard.
- Lexicographic sorting is
  - *fast*, even for very large tables.
  - *easy*: `sort` is a Unix staple.
Improving compression by sorting the table

- **RLE, BBC, WAH, EWAH** are order-sensitive: they compress *sorted* data better;
- But finding the *best* row ordering is NP-hard.
- Lexicographic sorting is
  - *fast*, even for very large tables.
  - *easy*: sort is a Unix staple.
- Substantial index-size reductions (often 2.5 times)
With $L$ bitmaps, you can represent $L$ values by mapping each value to **one bitmap**;

<table>
<thead>
<tr>
<th>1-of-$N$</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>cat</td>
</tr>
<tr>
<td>010000</td>
<td>dog</td>
</tr>
<tr>
<td>001000</td>
<td>dish</td>
</tr>
<tr>
<td>000100</td>
<td>fish</td>
</tr>
<tr>
<td>000010</td>
<td>cow</td>
</tr>
<tr>
<td>100000</td>
<td>cat</td>
</tr>
<tr>
<td>000001</td>
<td>pony</td>
</tr>
</tbody>
</table>
With \( L \) bitmaps, you can represent \( L \) values by mapping each value to one bitmap;

Alternatively, you can represent \( \binom{L}{2} = \frac{L(L - 1)}{2} \) values by mapping each value to a pair of bitmaps;

\[
\begin{array}{ccccccc}
1-of-N & 2-of-N \\
100000 & 1100 \\
010000 & 1010 \\
001000 & 1001 \\
000100 & 0110 \\
000010 & 0101 \\
100000 & 1100 \\
000001 & 0011 \\
\end{array}
\]
### $k$-of-$N$ encoding

<table>
<thead>
<tr>
<th>1-of-$N$</th>
<th>2-of-$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>1100</td>
</tr>
<tr>
<td>010000</td>
<td>1010</td>
</tr>
<tr>
<td>001000</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>0101</td>
</tr>
<tr>
<td>100000</td>
<td>1100</td>
</tr>
<tr>
<td>000001</td>
<td>0011</td>
</tr>
</tbody>
</table>

- With $L$ bitmaps, you can represent $L$ values by mapping each value to **one bitmap**;
- Alternatively, you can represent $\binom{L}{2} = \frac{L(L-1)}{2}$ values by mapping each value to a **pair of bitmaps**;
- More generally, you can represent $\binom{L}{k}$ values by mapping each value to a **$k$-tuple of bitmaps**;

With $L$ bitmaps, you can represent $L$ values by mapping each value to **one bitmap**;

Alternatively, you can represent $\binom{L}{2} = \frac{L(L-1)}{2}$ values by mapping each value to a **pair of bitmaps**;

More generally, you can represent $\binom{L}{k}$ values by mapping each value to a **$k$-tuple of bitmaps**;

At query time, you need to load $k$ bitmaps in a look-up for one value;

You trade query-time performance for fewer bitmaps;

Often, fewer bitmaps translates into a smaller index, created faster.
1-of-$N$ 2-of-$N$
100000 1100
010000 1010
001000 1001
000100 0110
000010 0101
100000 1100
000001 0011

- With $L$ bitmaps, you can represent $L$ values by mapping each value to one bitmap;
- Alternatively, you can represent $\binom{L}{2} = L(L-1)/2$ values by mapping each value to a pair of bitmaps;
- More generally, you can represent $\binom{L}{k}$ values by mapping each value to a $k$-tuple of bitmaps;
- At query time, you need to load $k$ bitmaps in a look-up for one value;
**$k$-of-$N$ encoding**

With $L$ bitmaps, you can represent $L$ values by mapping each value to **one bitmap**;

Alternatively, you can represent $\binom{L}{2} = \frac{L(L-1)}{2}$ values by mapping each value to a **pair of bitmaps**;

More generally, you can represent $\binom{L}{k}$ values by mapping each value to a **$k$-tuple of bitmaps**;

At query time, you need to load $k$ bitmaps in a look-up for one value;

You trade **query-time performance** for fewer bitmaps;

<table>
<thead>
<tr>
<th>1-of-$N$</th>
<th>2-of-$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>1100</td>
</tr>
<tr>
<td>010000</td>
<td>1010</td>
</tr>
<tr>
<td>001000</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>0101</td>
</tr>
<tr>
<td>100000</td>
<td>1100</td>
</tr>
<tr>
<td>000001</td>
<td>0011</td>
</tr>
</tbody>
</table>
**$k$-of-$N$ encoding**

- With $L$ bitmaps, you can represent $L$ values by mapping each value to a **one bitmap**;
- Alternatively, you can represent $\binom{L}{2} = \frac{L(L-1)}{2}$ values by mapping each value to a **pair of bitmaps**;
- More generally, you can represent $\binom{L}{k}$ values by mapping each value to a **$k$-tuple of bitmaps**;
- At query time, you need to load $k$ bitmaps in a look-up for one value;
- You trade **query-time performance** for **fewer bitmaps**;
- Often, **fewer bitmaps translates into a smaller index, created faster**.

<table>
<thead>
<tr>
<th>1-of-$N$</th>
<th>2-of-$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>1100</td>
</tr>
<tr>
<td>010000</td>
<td>1010</td>
</tr>
<tr>
<td>001000</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>0101</td>
</tr>
<tr>
<td>100000</td>
<td>1100</td>
</tr>
<tr>
<td>000001</td>
<td>0011</td>
</tr>
</tbody>
</table>
Gray-code order, when $k > 1$

- A Gray Code (GC) minimizes bit transitions: 00, 01, 11, 10
Gray-code order, when $k > 1$

- A **Gray Code** (GC) minimizes bit transitions: 00, 01, 11, 10
- Pinar et al. propose to sort whole index by $<_{\text{GC}}$, Gray-code ordering. Practical?
A Gray Code (GC) minimizes bit transitions: 00, 01, \textbf{11}, 10

Pinar et al. propose to sort whole index by $<_{\text{GC}}$, Gray-code ordering. Practical?

We contribute an easy/fast way to achieve GC-like results using lexicographic sort.
A **Gray Code** (GC) minimizes bit transitions: 00, 01, 11, 10

Pinar et al. propose to sort whole index by $<_{GC}$, Gray-code ordering. Practical?

We contribute an easy/fast way to achieve GC-like results using lexicographic sort.

Empirical improvement in index size: typically 0–4%.
Gray-code order, when $k > 1$

- A **Gray Code** (GC) minimizes bit transitions: 00, 01, 11, 10
- Pinar et al. propose to sort whole index by $<_{GC}$, Gray-code ordering. Practical?
- We contribute an easy/fast way to achieve GC-like results using lexicographic sort.
- Empirical improvement in index size: typically 0–4%.
- Paper has details.
Experimental environment

- Mac Pro with 2 dual-core CPUs 2 GiB RAM (no thrashing)
- GNU GCC 4.0.2 (C++)—32-bit binaries

Source code under GPL:
http://code.google.com/p/lemurbitmapindex/ (Linux and MacOS)
Experimental environment

- Mac Pro with 2 dual-core CPUs 2 GiB RAM (no thrashing)
- GNU GCC 4.0.2 (C++)—32-bit binaries
- **Source code under GPL:**
  http://code.google.com/p/lemurbitmapindex/
  (Linux and MacOS)
Experimental environment

- Mac Pro with 2 dual-core CPUs 2 GiB RAM (no thrashing)
- GNU GCC 4.0.2 (C++)—32-bit binaries
- **Source code under GPL:**
  (Linux and MacOS)
- Mix of real and synthetic data:
  1. up to 877 M rows, 22 GB, 4 M attributes.
  2. experiments using 4–10 columns
If $k > 1$, bitmaps are denser and a query processes $k$ of them;
If $k > 1$, bitmaps are denser and a query processes $k$ of them; cost can grow with $n_j^{(k-1)/k}$ (big jump from 1 to 2).
1. If $k > 1$, bitmaps are denser and a query processes $k$ of them; cost can grow with $n^{(k-1)/k}$ (big jump from 1 to 2).
2. If $k > 1$, we (usually) get smaller indexes.
1. If $k > 1$, bitmaps are denser and a query processes $k$ of them; cost can grow with $n_i^{(k-1)/k}$ (big jump from 1 to 2).

2. If $k > 1$, we (usually) get smaller indexes. Potentially high query-time penalty for $k > 1$, at best modest space gains.
1. If $k > 1$, bitmaps are denser and a query processes $k$ of them; cost can grow with $n^{(k-1)/k}$ (big jump from 1 to 2).

2. If $k > 1$, we (usually) get smaller indexes.

Potentially high query-time penalty for $k > 1$, at best modest space gains.

**Default choice of $k$**

Our index-construction algorithm handles the extremely sparse ($k = 1$) indexes nicely. $k = 1$ looks like a good choice.
The first column(s) gain more from the sort (column 1 is primary sort key);
When sorting, column order matters

- The first column(s) gain more from the sort (column 1 is primary sort key);
- Conceptually, we may wish to reorder columns, eg swap columns 1 & 3.
When sorting, column order matters

- The first column(s) gain more from the sort (column 1 is primary sort key);
- Conceptually, we may wish to reorder columns, eg swap columns 1 & 3.
- Column order is crucial!

Netflix: 24 column orderings

index size
column permutation
1-of-N encoding
4-of-N encoding

Owen Kaser¹, Daniel Lemire², Kamel Aouiche²

Histogram-Aware Sorting for Enhanced Word-Aligned Compression in Bitmap Indexes
When sorting, column order matters

- The first column(s) gain more from the sort (column 1 is primary sort key);
- Conceptually, we may wish to reorder columns, eg swap columns 1 & 3.
- Column order is crucial!
- Finding the best ordering quickly remains open.

Netflix: 24 column orderings

![Graph showing index size vs column permutation for Netflix with 1-of-N and 4-of-N encoding]
Progress toward choosing column order

- Paper models “gain” of putting a given column first.
- Idea: order columns greedily (by max gain).
Paper models “gain” of putting a given column first.

Idea: order columns greedily (by max gain).

Experimentally, this approach is not promising: the best orderings don’t seem to depend on gain.
Paper models “gain” of putting a given column first.

Idea: order columns greedily (by max gain).

Experimentally, this approach is not promising: the best orderings don’t seem to depend on gain.

Factors:

- skews of columns
- number of distinct values
- \( k \)
- density of column’s bitmaps
For 1-of-$N$ bitmaps, a density-based approach was okay:
For 1-of-$N$ bitmaps, a density-based approach was okay:

**Ordering rule, $k = 1$: “sparse but not too sparse”**

Order columns by decreasing

$$\min \left( \frac{1}{n_i}, \frac{1 - 1/n_i}{4w - 1} \right),$$

where

- $n_i \rightarrow$ the number of distinct values in column $i$,
- $w \rightarrow$ the word size.
What usually works for dimension ordering?: $k=1$

For 1-of-$N$ bitmaps, a density-based approach was okay:

**Ordering rule, $k = 1$: “sparse but not too sparse”**

Order columns by decreasing

$$\min \left( \frac{1}{n_i}, \frac{1 - 1/n_i}{4w - 1} \right),$$

where

- $n_i \rightarrow$ the number of distinct values in column $i$,
- $w \rightarrow$ the word size.

See 30–40% size reduction, merely knowing dimension sizes ($n_i$).
What usually works for dimension ordering?: \( k=1 \)

For 1-of-\( N \) bitmaps, a density-based approach was okay:

**Ordering rule, \( k = 1 \): “sparse but not too sparse”**

Order columns by decreasing

\[
\min \left( \frac{1}{n_i}, \frac{1 - 1/n_i}{4w - 1} \right)
\]

- \( n_i \rightarrow \) the number of distinct values in column \( i \),
- \( w \rightarrow \) the word size.

See 30–40% size reduction, merely knowing dimension sizes (\( n_i \)).
See also [Canahuate et al., 2006] for related work.
What usually works for dimension ordering?: $k=1$

For 1-of-$N$ bitmaps, a density-based approach was okay:

**Ordering rule, $k = 1$**: “sparse but not too sparse”

Order columns by decreasing

$$\min\left(\frac{1}{n_i}, \frac{1 - 1/n_i}{4w - 1}\right),$$

where

- $n_i \to$ the number of distinct values in column $i$,
- $w \to$ the word size.

See 30–40% size reduction, merely knowing dimension sizes ($n_i$).
See also [Canahuate et al., 2006] for related work.
Situation worse for $k > 1$. [Details]
Future directions

- Need better **mathematical modelling** of bitmap compressed size in sorted tables;
Future directions

- Need better **mathematical modelling** of bitmap compressed size in sorted tables;
- Study the effect of **word length** (16, 32, 64, 128 bits);
Future directions

- Need better **mathematical modelling** of bitmap compressed size in sorted tables;
- Study the effect of **word length** (16, 32, 64, 128 bits);
- Apply to **Column**-oriented DBMS [Stonebraker et al., 2005];
Future directions

- Need better **mathematical modelling** of bitmap compressed size in sorted tables;
- Study the effect of **word length** (16, 32, 64, 128 bits);
- Apply to Column-oriented DBMS [Stonebraker et al., 2005];
- Consider encodings that can efficiently support range queries [Chan and Ioannidis, 1999].


Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S.,

Owen Kaser¹, Daniel Lemire², Kamel Aouiche²

Histogram-Aware Sorting for Enhanced Word-Aligned Compression

Density formula \((n_i \rightarrow \sqrt[k]{n_i})\) recommends poorly when \(k > 1\). Our experiments on synthetic data give some guidance:
Density formula ($n_i \rightarrow \sqrt{k n_i}$) recommends poorly when $k > 1$. Our experiments on synthetic data give some guidance:

When $k > 1$, order columns by

1. descending skew
2. descending size

(And do the reverse when $k = 1$.)
What usually works for dimension ordering? $k > 1$

Density formula ($n_i \rightarrow \sqrt[k]{n_i}$) recommends poorly when $k > 1$. Our experiments on synthetic data give some guidance:

When $k > 1$, order columns by

1. descending skew
2. descending size

(And do the reverse when $k = 1$.)

Open issues, $k > 1$

1. How do we balance skew & size factors?
What usually works for dimension ordering?: $k > 1$

Density formula ($n_i \rightarrow \sqrt[k]{n_i}$) recommends poorly when $k > 1$. Our experiments on synthetic data give some guidance:

When $k > 1$, order columns by

1. descending skew
2. descending size

(And do the reverse when $k = 1$.)

Open issues, $k > 1$

1. How do we balance skew & size factors?
2. What other properties of the histograms are needed?
Gray-code order

<table>
<thead>
<tr>
<th>Lex. order</th>
<th>Gray-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

- **Gray-code (GC)** order is an alternative to lexicographical order (defined only for bit arrays);

**May improve compression more than lex. sort** ($k > 1$);

**[Pinar et al., 2005]** process a materialized bitmap index.

**Slow**, if uncompressed index does not fit in RAM.

GC order is not supported by DBMSes or Unix utilities.
Gray-code order

<table>
<thead>
<tr>
<th>Lex. order</th>
<th>Gray-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

- **Gray-code** (GC) order is an alternative to lexicographical order (defined only for bit arrays);
- May improve compression more than lexic. sort \((k > 1)\);
<table>
<thead>
<tr>
<th>Lex. order</th>
<th>Gray-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

- **Gray-code** (GC) order is an alternative to lexicographical order (defined only for bit arrays);
- May improve compression more than lex. sort \((k > 1)\);
- [Pinar et al., 2005] process a materialized bitmap index.
Gray-code order

<table>
<thead>
<tr>
<th>Lex. order</th>
<th>Gray-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

- **Gray-code** (GC) order is an alternative to lexicographical order (defined only for bit arrays);
- May improve compression more than lex. sort ($k > 1$);
- [Pinar et al., 2005] process a materialized bitmap index.
- Slow, if uncompressed index does not fit in RAM.
Gray-code order

<table>
<thead>
<tr>
<th>Lex. order</th>
<th>Gray-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

- **Gray-code** (GC) order is an alternative to lexicographical order (defined only for bit arrays);
- May improve compression more than lex. sort \((k > 1)\);
- [Pinar et al., 2005] process a materialized bitmap index.
- Slow, if uncompressed index does not fit in RAM.
- GC order is not supported by DBMSes or Unix utilities.
Gray-code sorting, cheaply

Size improvement is small (usually $< 4\%$), but it’s essentially free:

1. What Pinar et al. do: expensive GC sort after encoding
   eg: [Tax, Cat, Girl, Cat] → sort([1100, 0110, 1001, 0110]);

2. Instead, sort the table lexicographically;
   eg: [Cat, Cat, Girl, Tax] → [Cat, Cat, Girl, Tax];

3. Map ordered values to $k$-tuples of bitmaps ordered as Gray codes: Cat: 0011, Dog: 0110, Girl: 0101, Tax: 1100;
   Lex ascending sequence: Cat, Dog, Girl, Tax.
   GC ascending sequence: 0011, 0110, 0101, 1100 for codes
   eg: [Cat, Cat, Girl, Tax] → [0011, 0011, 0101, 1100] (generates a GC-sorted result without expensive GC sorting).

4. Easily extended for $> 1$ columns.
   In our tests, this is as good as a Gray-code bitmap index [Pinar et al., 2005], but technically much easier.
Gray-code sorting, cheaply

Size improvement is small (usually < 4%), but it’s essentially free:

1. What Pinar et al. do: expensive GC sort after encoding
   eg: [Tax, Cat, Girl, Cat] → \text{sort}([1100, 0110, 1001, 0110]);

2. Instead, sort the table lexicographically;
   eg: [Tax, Cat, Girl, Cat] → [Cat, Cat, Girl, Tax]

Map ordered values to $k$-tuples of bitmaps ordered as Gray codes: Cat: 0011, Dog: 0110, Girl: 0101, Tax: 1100; Lex ascending sequence: Cat, Dog, Girl, Tax.

GC ascending sequence: 0011, 0110, 0101, 1100 for codes
eg: [Cat, Cat, Girl, Tax] → [0011, 0011, 0101, 1100] (generates a GC-sorted result without expensive GC sorting).

Easily extended for $>1$ columns. In our tests, this is as good as a Gray-code bitmap index [Pinar et al., 2005], but technically much easier.
Gray-code sorting, cheaply

Size improvement is small (usually < 4%), but it’s essentially free:

1. What Pinar et al. do: expensive GC sort after encoding
   eg: [Tax, Cat, Girl, Cat] → `sort([1100, 0110, 1001, 0110])`;

2. Instead, sort the table lexicographically;
   eg: [Tax, Cat, Girl, Cat] → [Cat, Cat, Girl, Tax]

3. Map ordered values to $k$-tuples of bitmaps ordered as Gray codes:
   Cat: 0011, Dog: 0110, Girl: 0101, Tax: 1100;

   Lex ascending sequence: Cat, Dog, Girl, Tax.
   GC ascending sequence: 0011, 0110, 0101, 1100 for codes
Gray-code sorting, cheaply

Size improvement is small (usually $< 4\%$), but it’s essentially free:

1. What Pinar et al. do: expensive GC sort after encoding
eg: [Tax, Cat, Girl, Cat] → sort([1100, 0110, 1001, 0110]);

2. Instead, sort the table lexicographically;
   eg: [Tax, Cat, Girl, Cat] → [Cat, Cat, Girl, Tax]

3. Map ordered values to $k$-tuples of bitmaps ordered as Gray codes:
   Cat: 0011, Dog: 0110, Girl: 0101, Tax: 1100;
   Lex ascending sequence: Cat, Dog, Girl, Tax.
   GC ascending sequence: 0011, 0110, 0101, 1100 for codes
   eg: [Cat, Cat, Girl, Tax] → [0011, 0011, 0101, 1100]
   (generates a GC-sorted result without expensive GC sorting).
Gray-code sorting, cheaply

Size improvement is small (usually $< 4\%$), but it’s essentially free:

1. What Pinar et al. do: expensive GC sort after encoding
   eg: [Tax, Cat, Girl, Cat] $\rightarrow$ sort([1100, 0110, 1001, 0110]);

2. Instead, sort the table lexicographically;
   eg: [Tax, Cat, Girl, Cat] $\rightarrow$ [Cat, Cat, Girl, Tax]

3. Map **ordered values** to $k$-tuples of bitmaps ordered as Gray codes:

   Lex ascending sequence: Cat, Dog, Girl, Tax.
   GC ascending sequence: 0011, 0110, 0101, 1100 for codes
   eg: [Cat, Cat, Girl, Tax] $\rightarrow$ [**0011**, **0011**, **0101**, **1100**]
   (generates a GC-sorted result without expensive GC sorting).

4. Easily extended for $> 1$ columns.
Gray-code sorting, cheaply

Size improvement is small (usually < 4%), but it’s essentially free:

1. What Pinar et al. do: expensive GC sort after encoding
   eg: [Tax, Cat, Girl, Cat] → sort([1100, 0110, 1001, 0110]);

2. Instead, sort the table lexicographically;
   eg: [Tax, Cat, Girl, Cat] → [Cat, Cat, Girl, Tax]

3. Map ordered values to $k$-tuples of bitmaps ordered as Gray codes: Cat: 0011, Dog: 0110, Girl: 0101, Tax: 1100;
   Lex ascending sequence: Cat, Dog, Girl, Tax.
   GC ascending sequence: 0011, 0110, 0101, 1100 for codes
   eg: [Cat, Cat, Girl, Tax] → [0011, 0011, 0101, 1100]
   (generates a GC-sorted result without expensive GC sorting).

4. Easily extended for $>1$ columns.

In our tests, this is as good as a Gray-code bitmap index sort [Pinar et al., 2005], but technically much easier.