Bitmap Index Design Choices and Their Performance Implications

Elizabeth O'Neil, Patrick O'Neil and Kesheng Wu

International Database Engineering and Application Symposium
IDEAS(2007)

Elizabeth O'Neil, Patrick O'Neil

University of Massachusetts at Boston

Produced the **first** commercial implementation of bitmap indexes MODEL 204 in the DBMS in the early 1980s. This work was first published in 1987.

the author of the database textbook ***Database Principles, Programming, and Performance***

Jhon Wu (Kesheng Wu)

Lawrence Berkeley National Laboratory

Author of Fastbit, a NoSQL database using bitmap index entirely;

Own WAH algorithm, a compression algorithm for bitmap index;

Lawrence Berkeley National Laboratory

Libpcap **tracerout** **Tpcdump.**

a major international center for physics research

Outlines

1. Background
   No definitive design

2. Purpose
   Investigate an efficient design on modern processors

3. Methodology
   By comparison between two Prototypes

4. Two Prototype Databases
   Fastbit & RIDbit

5. Theoretical analysis

6. Experiments
   Set Query Benchmark: 5 problems

7. Conclusion

Background

What's a bitmap
(basic case)

There is a table with **column** named **X**, which ranges from 0 to 3.
The ***column cardinality*** of **X** is 4.

RID	**X**	B_0	B_1	B_2	B_3
0	2	0	0	1	0
1	1	0	1	0	0
2	3	0	0	0	1
3	0	1	0	0	0
4	3	0	0	0	1
5	1	0	1	0	0
6	0	1	0	0	0
7	0	1	0	0	0
8	2	0	0	1	0

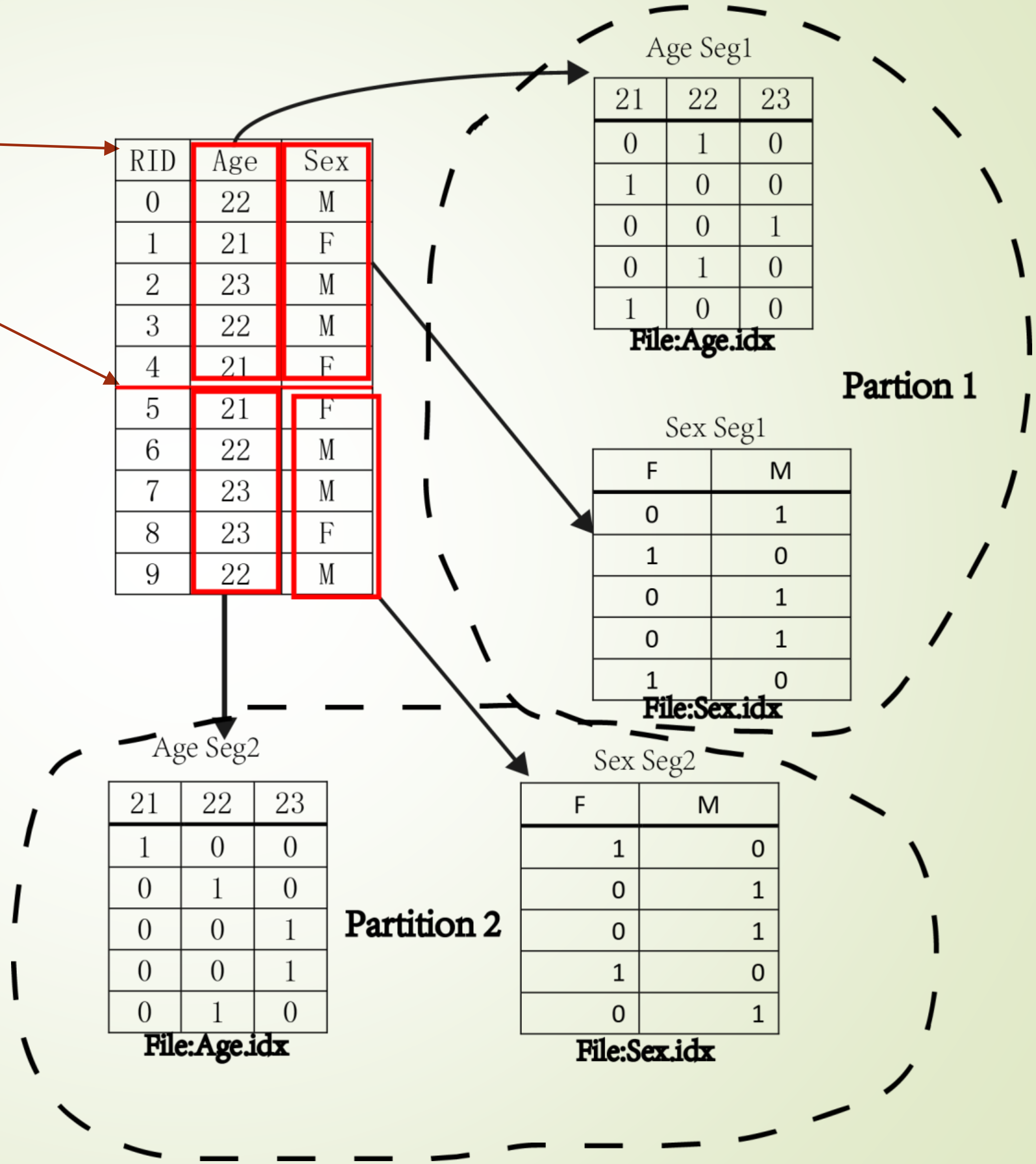
Background

Current implementations

MODEL 204 by E. O'neil & P O'neil

Fastbit

- Organize data as **tables**
- Horizontal partitioned
- Indexes of columns are stored in separated files
- Index for fixed-size columns
- WAH compression
- Generate whole bitmap for a partition in memory
- Bit vectors for different columns stored sequentially



Index File (*.idx) Structure

```
{
    Headers;  // no help for understanding, just ignore it
    starts[];     // start position of each bit sequence
    values[];   //column values occur in source data
    bitsequences[]; // bitmap stored column by column
}
```

The source data can be retrieved from index file, which avoid reading source files.

Unfixed-size data type like String will be transformed into integers by mapping. This is will get another file involved into retrieving.

RIDbit

Source Data Storage

Based on disk page(32Kbit/4KB), which means segmentation

For a table **T** with **N** records: $\boldsymbol{T} = \{\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{r_3} \ldots, \boldsymbol{r_N}\}$
Assign each record $\boldsymbol{r_i}$ a ***row number*** $\boldsymbol{m}[\boldsymbol{r_i}] \in \{\boldsymbol{0}, \ldots, \boldsymbol{M}\}, \boldsymbol{M} > \boldsymbol{N}$

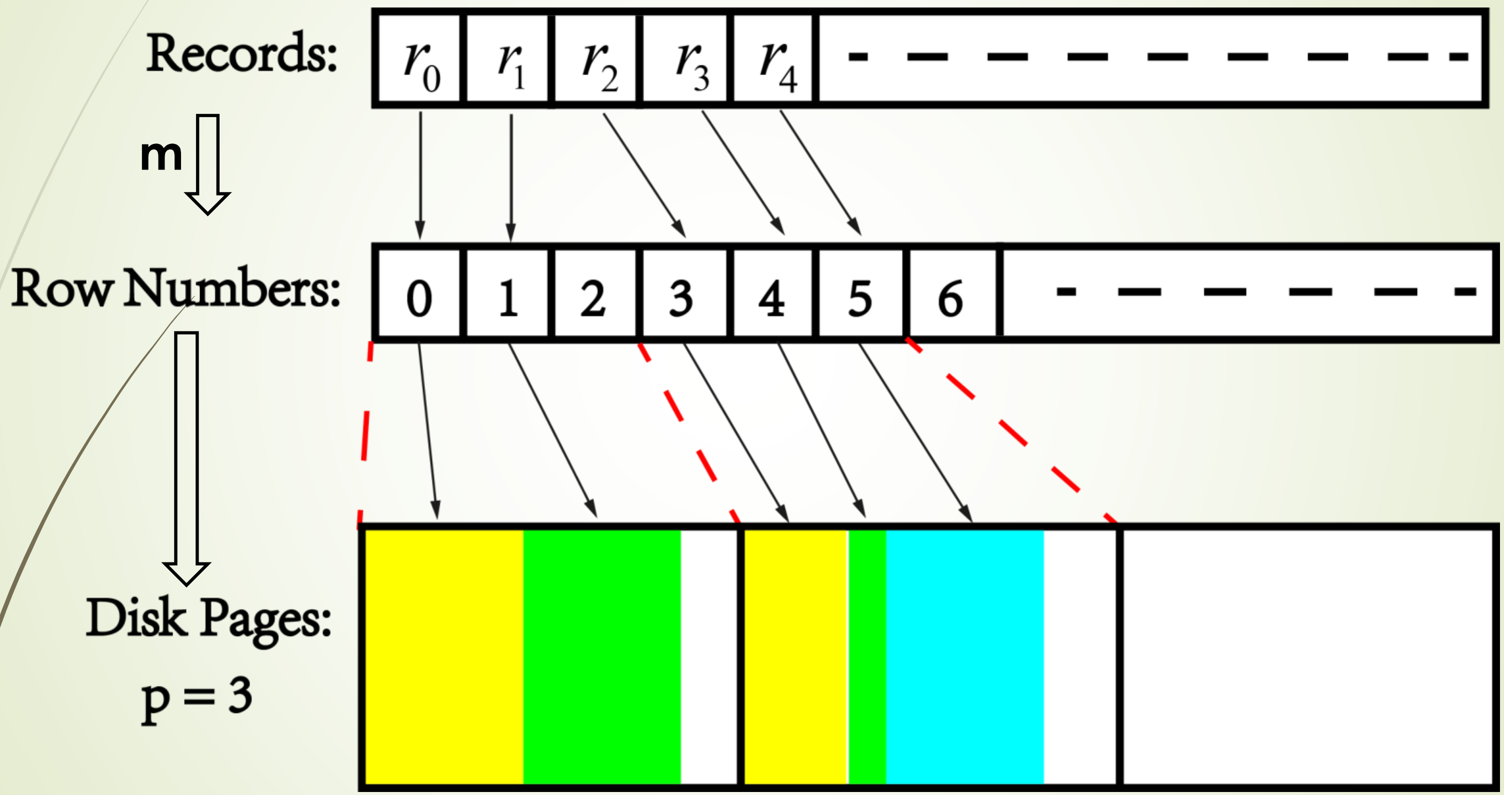
Let **p** *be the* ***page capacity***, which means that one disk page may contain up to p records.

Then $m[r_i]/p$ is the pageID of the disk page containing r_i and $m[r_i]\%p$ is the slotID of r_i.

r_i is stored in row-oriented style (horizontal data organization)

RIDbit

Source Data Storage



RIDbit

Indexes Storage Strategy

Based on disk page (32Kbit/4KB), which means segmentation

Column values are organized as key-values in B-Tree indexes. The bitmap corresponding to different values are linked to the B-Tree.

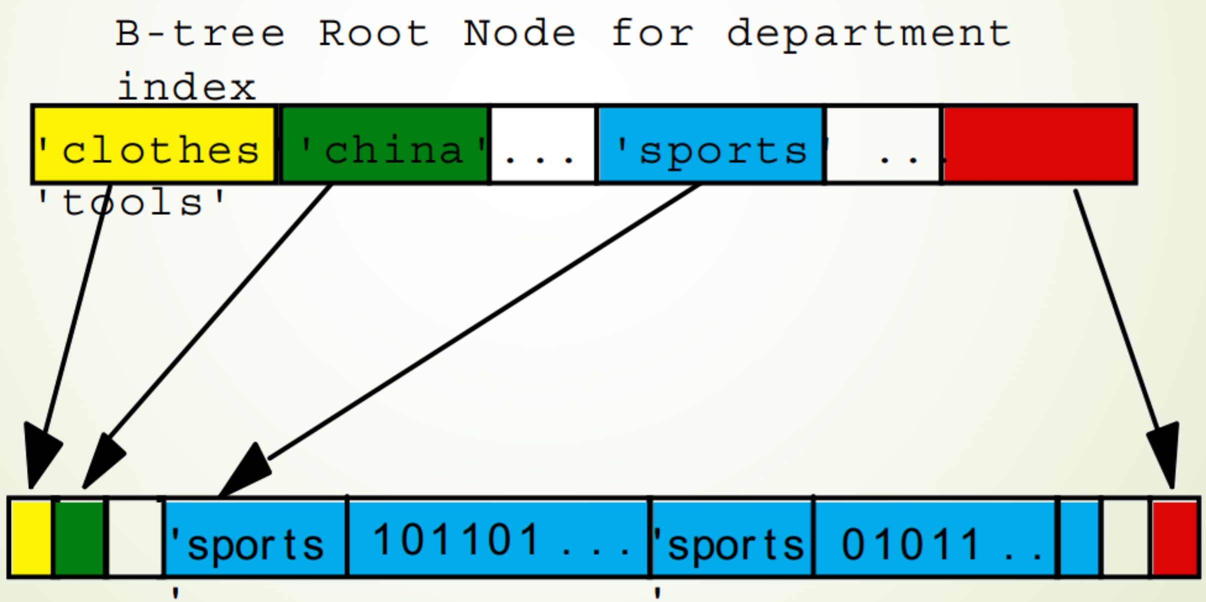


Figure 1: A RIDBit Index on department, a column of the SALES table.

RIDbit

Indexes Storage Strategy

RIDbit is capable of three type of indexes:
For an integer set like **{0,3,4,5,7,10,12,19}** partitioned in segments containing **8** records.

1. Segmented verbatim bitmap (No Compression)
   - Seg0: 10011101
   - Seg1: 00101000
   - Seg2: 00010000
2. Segmented-relative RID-list
   - Seg0: {0, 3, 4, 5, 7}
   - Seg1: {2, 4}
   - Seg2: {3}
3. Full-size RID-list
   - {0,3,4,5,7,10,12,19}

Fastbit VS. RIDbit

Table 2: Key differences between RIDBit and FastBit.

	FastBit	RIDBit
Table layout	Vertical storage (columns stored separately)	N-ary storage (columns stored together in row)
Index layout	Arrays of bitmaps	B-tree keyed on keyvalues (improved in project)
Bitmap layout	Continuous	Horizontally partitioned into 32K-bit Segments
Compression	Word-Aligned Hybrid compression	Sparse bitmap converted to RID-list

Theoretical analysis

on index size per row

Random Uniform Distribution

C is the column cardinality.
w = 32, indicating 32bit WAH compression
N is the number of records

Fastbit:

$(w/w\text{-}1)\ C$	small C
$(Cw/(w-1))(1 - e^{-(2w-2)/C}) \approx 2w$	$1 \ll C \ll N$
$5w = 160$	$C \sim N$

RIDbit:

C	in bitmap format, when C is small
16	in Segmented RID-list, when $C < k * 32{,}000$
64	in Full-size RID-list, $C > k * 32{,}000$

Theoretical analysis

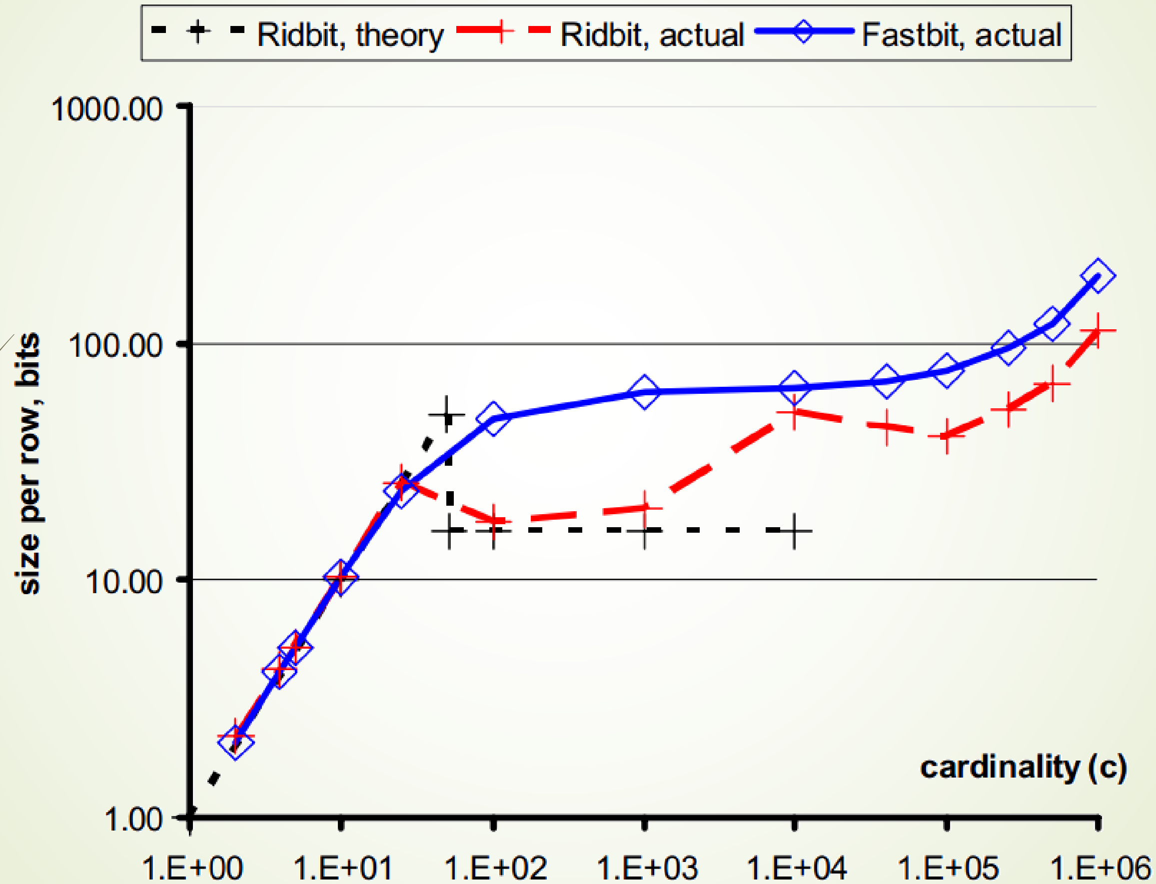


Figure 2: Index sizes vs. column cardinality.

Set Query Benchmark

Table 4: Set Query Benchmark columns and their column cardinalities.

Name	Cardinality
KSEQ	1,000,000
K500K	500,000
K250K	250,000
K100K	100,000
K40K	40,000
K10K	10,000
K1K	1,000
K100	100
K25	25
K10	10
K5	5
K4	4
K2	2

$Q1. select\ count(*)\ from\ BENCH$
$\quad where\ KN = 2;$

$Q2A. select\ count(*)\ from\ BENCH$
$\quad where\ KN = 2\ and\ KN = 3;$

$Q2B. select\ count(*)\ from\ BENCH$
$\quad where\ KN = 2\ and\ not\ KN = 3;$

$Q3A0. select\ sum(K1K)\ from\ BENCK$
$\quad where\ KSEQ\ between\ 400000\ and\ 500000$
$\quad\quad and\ KN = 3;$

$Q3B0. select\ sum(K1K)\ from\ BENCH$
$\quad where\ (KSEQ\ between\ 400000\ and\ 410000$
$\quad\quad or\ KSEQ\ between\ 420000\ and\ 430000$
$\quad\quad or\ KSEQ\ between\ 440000\ and\ 450000$
$\quad\quad or\ KSEQ\ between\ 460000\ and\ 470000$
$\quad\quad or\ KSEQ\ between\ 480000\ and\ 500000)$
$\quad\quad and\ KN = 3;$

Set Query Benchmark

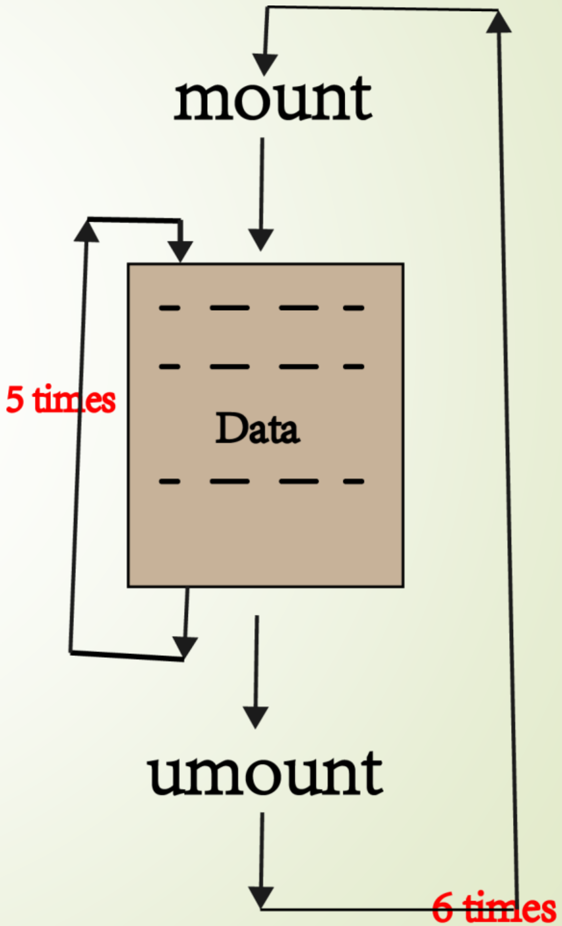
$$Q4A. selectK\ SEQ, K500K, from\ BENCH$$
$$where\ (i) \sim (i+2)$$
$$Q4B. selectK\ SEQ, K500K, from\ BENCH$$
$$where\ (i) \sim (i+4)$$
$$Q5. select\ KN1, KN2, count(*), group\ by\ KN1, KN2;$$

Table 5: Range conditions used for Q4.

(1)	K2 = 1
(2)	K100 > 80
(3)	K10K between 2000 and 3000
(4)	K5 = 3
(5)	K25 in (11, 19)
(6)	K4 = 3
(7)	K100 < 41
(8)	K1K between 850 and 950
(9)	K10 = 7
(10)	K25 in (3, 4)

Experiment setup

- Mount/Umount before/after experiment
- Copy data source to get five sets
- Repeat whole process for 6 times (sum up to 30 runs for a query)



Conclusions

- Vertical data organization is better
- Clustered index organization is better

Table 6: Information about the test systems.

	CPU		disk		
	Type	Clock (GHz)	Type	Latency (ms)	Speed (MB/s)
HDA	Pentium 4	2.2	EIDE	7.6	38.7
MD0	Pentium 4	2.2	Software RAID0 (2 disks)	9.4	58.8
SDA	Pentium 4	2.8	Hardware RAID0 (4 disks)	15.8	62.2
SDB	PowerPC 5	1.6	SCSI	8.3	54.4

Index Building

Table 7: Total index sizes (MB) and the time (in seconds) needed to build them.

		RIDBit	FastBit
	Size	64.2 MB	93.5b MB
time	**HDA**	75.7 sec	21.7 sec
	MD0	8.3 sec	27.2 sec
	SDA	3.5 sec	34.8 sec
	SDB	4.0 sec	41.7 sec

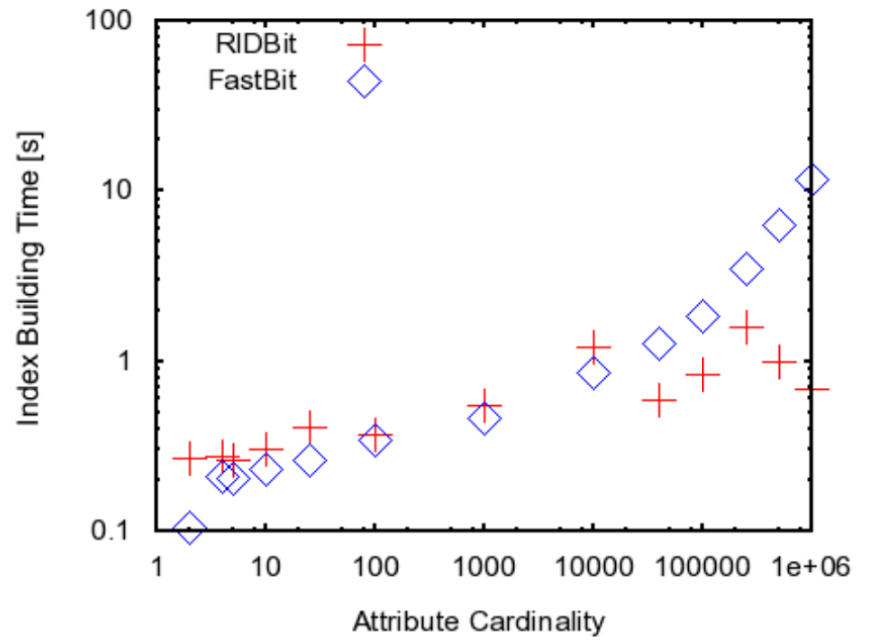


Figure 3: Time (in seconds) required to build each individual index on system MD0.

Index-Only Query

Table 8: Total elapsed time (seconds) to answer the count queries on four test systems.

	HDA		**MD0**		**SDA**		**SDB**	
	RIDBit	FastBit	RIDBit	FastBit	RIDBit	FastBit	RIDBit	FastBit
Q1	0.39	0.23	0.50	0.25	0.34	0.26	0.52	0.22
Q2A	0.74	0.42	0.68	0.51	0.50	0.47	0.85	0.53
Q2B	0.71	0.42	0.66	0.49	0.53	0.46	0.88	0.52
Q3A0	2.28	2.18	2.00	1.91	1.79	1.73	1.97	2.06
Q3B0	2.08	2.46	1.76	1.90	1.49	1.41	1.87	1.81
Q4A0	1.39	0.94	1.22	0.83	0.97	0.77	2.10	1.03
Q4B0	2.20	1.46	1.75	1.31	1.67	1.21	2.98	1.65
Q5	1.13	1.44	1.09	1.46	0.81	1.21	1.03	1.50
Total	10.92	9.55	9.66	8.66	8.10	7.52	12.20	9.32

Clustered Index

Reading sequentially

MD0

Table 9: Number of disk sectors (in thousands) needed to answer count queries.

	RIDBit	FastBit	MD0 RIDBit	MD0 FastBit
Q1	10.7	4.9	0.50	0.25
Q2A	15.0	9.2	0.68	0.51
Q2B	15.0	9.2	0.66	0.49
Q3A0	52.0	64.4	2.00	1.91
Q3B0	34.4	48.8	1.76	1.90
Q4A0	27.8	35.9	1.22	0.83
Q4B0	41.3	56.5	1.75	1.31
Q5	23.0	27.4	1.09	1.46
Total	219.2	256.3	9.66	8.66

Index-Only Query

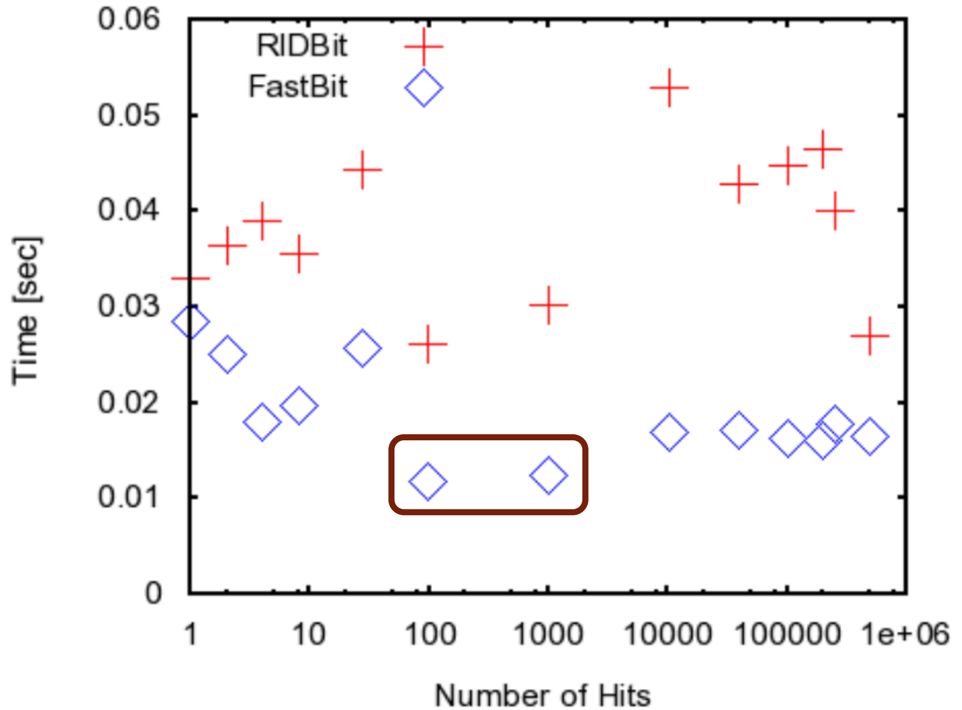


Figure 4: Elapsed time (seconds) to answer Q1 on MD0.

Table 10 Total CPU time (seconds) to answer count queries on MD0.

	RIDBit	FastBit
Q1	0.016	0.045
Q2A	0.028	0.059
Q2B	0.038	0.061
Q3A0	0.500	0.672
Q3B0	0.307	0.521
Q4A0	0.137	0.111
Q4B0	0.192	0.165
Q5	0.701	0.795
Total	1.919	2.429

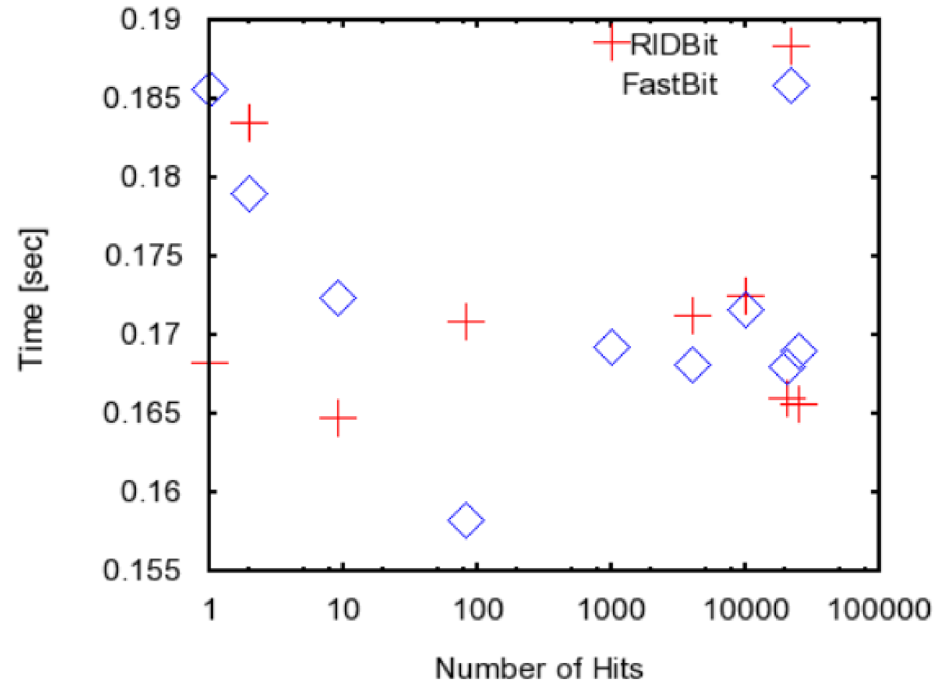


Figure 5 Elapsed Time (seconds) to answer Q3A0 on MD0.

Data Retrieval Query

Q3

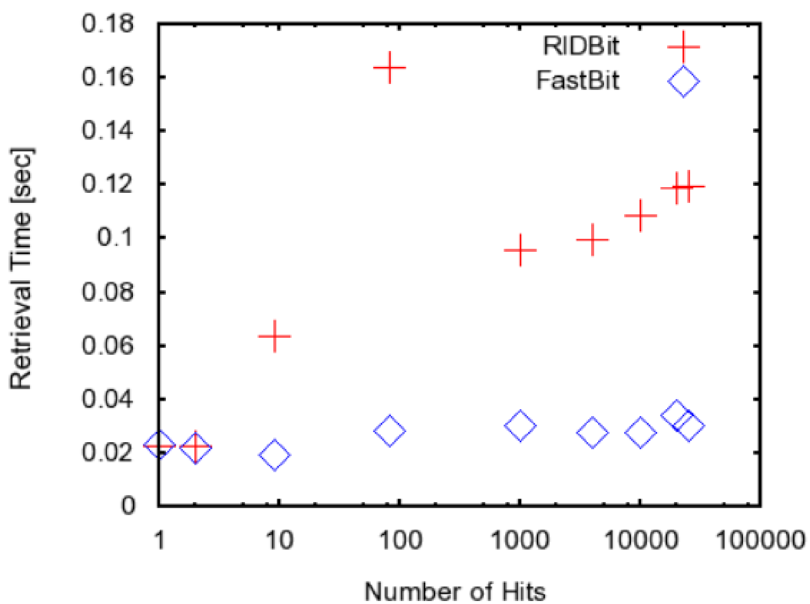


Figure 6 Time spent to retrieve the selected records to answer Q3A on MD0.

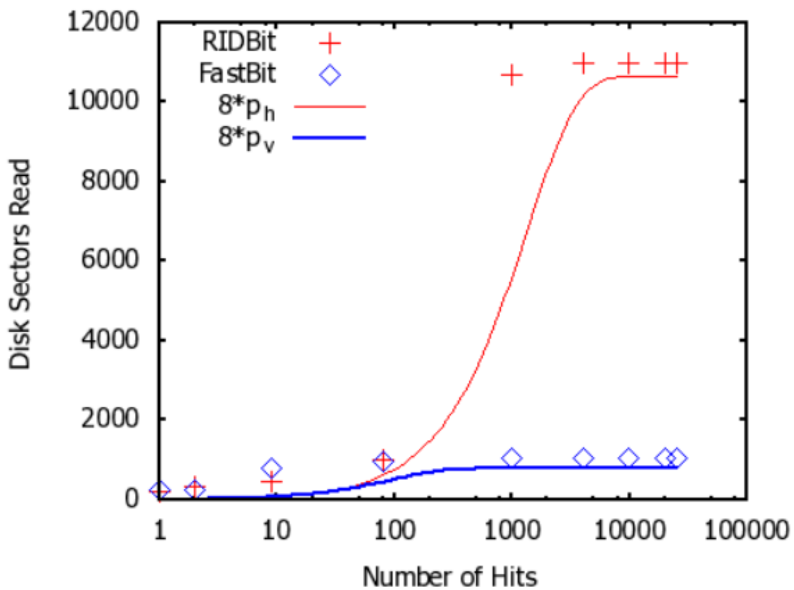


Figure 7 Number of disk sectors accessed to retrieve the records for Q3A.

Time VS. Columns

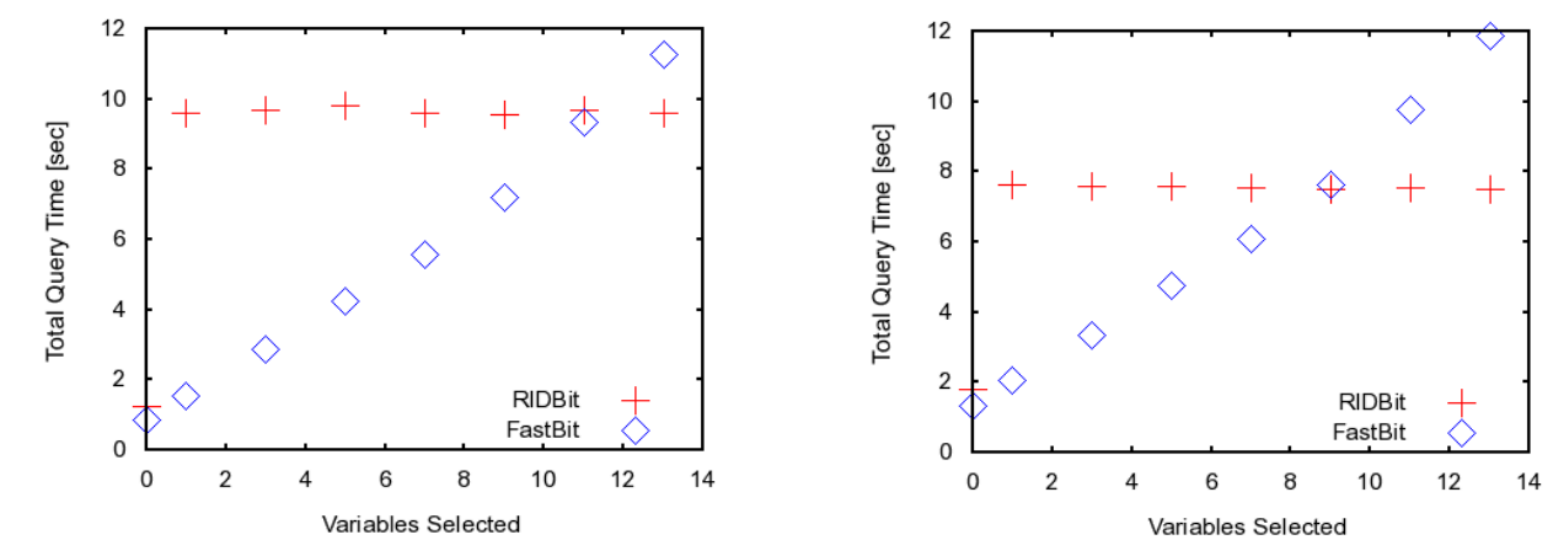


Figure 8 Total time used to answer Q4A on MD0.

Figure 9 Total time used to answer Q4B on MD0.

Conclusions

- Vertical data organization is better

  when only small number of columns get involved, which is the case usually.

- Clustered index organization is better

  Which avoid the delay brought by random access.

- Modifications for modern processors are needed
  - Branch-avoiding C code for compression algorithm
  - Optimization for sequential scan

My Work: New compression algorithm

- Adaptive Position List WAH algorithm (APLWAH)
- Further compression based on WAH
- Only one extra compression pattern added
- Adapt to 64/32 bit system

Modifications

- Apply APLWAH to Fastbit (Finished)
- Apply APLWAH to Druid, a distributed real-time database (Proceeding)
- Apply APLWAH to Lucene, an open source search software from Apache. (Starting)