How to index millions of integers per second with bitmap indexes

Lorenzo Vannucci

The aims of this work

We want to create a new type of index for “equality-query” (A = ‘x’) on integer attributes that:

- Allow to index millions of integers per second.

- Allow to perform a query efficiently on a billion-rows database.

- Requires a limited space on persistent storage.

Bitmap Indexing

Bitmap indexes have traditionally been considered to work well for low-cardinality columns, which have a modest number of distinct values. The simplest and most common method of bitmap indexing on attribute A with L cardinality associates a bitmap with every value V then the Vth bitmap represent the predicate A = V.

For each value inserted is set the corresponding bitmap.

	b0	b1	b2	b3	b4	b5	b6	b7
3	0	0	0	1	0	0	0	0
5	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0
2	0	0	1	0	0	0	0	0
7	0	0	0	0	0	0	0	1
	Bitmap	Bitmap	Bitmap	Bitmap	Bitmap	Bitmap	Bitmap	Bitmap

In this example
| A | = L = 8

The size of a bitmap index

This approach ensures an efficient solution in insertion (only one bit is set for each value inserted) and for performing search (is sufficient get the position of 1 bits set of the bitmap that corresponds to the sought value) but on
high-cardinality attributes the size of a bitmap index increase dramatically.

To reduce the cost of a bitmap index we have created a new bitmap compression exclusively optimized to create an index on a high-cardinality attributes using one bitmap for each distinct values.

Properties of a bitmap index

The fundamental property of a bitmap index composed from L bitmap is that for each row there is only one bitmap set and then for N rows there are N bits = 1 independently of the size of L. This implies that for L bitmaps and N rows there are in total (L-1) * N bits = 0 and N bits = 1.

In this scenario to minimize the size of a bitmap index (composed from L bitmap) is very important compress 0 bit sequences in efficient mode of each bitmap.

Only Zero Bytes Compressed

Only Zero Bytes Compressed (OZBC) is a word-aligned-hybrid compressed bitmap designed exclusively to create a bitmap indexes on high-cardinality attributes and provides bitwise logical operation in running time complexity proportional to the compressed bitmap size.

OZBC is implemented in C++ code on github and released on Gnu Lesser General Public Licensev3.0.

https://github.com/uccidibuti/OZBCBitmap .

The size of a OZBC Bitmap Index : 1

We can prove that in the worse case the size of a OZBC Bitmap Index that indexing an attribute A with L cardinality and N values inserted is:

(2 * log_2(L) * N) / 8 Bytes

+

L * 8 Bytes

1) I.e. :

the cost of a OZBC bitmap index on a int16 column (L = 2^16) with N = 10,000,000 values inserted is:

(2 * log_2(2^16) * 10M) / 8 + 2^19 = 40,5MBytes

TWICE THE COLUMN SIZE

The size of a OZBC Bitmap Index : 2

We can prove that in the worse case the size of a OZBC Bitmap Index that indexing an attribute A with L cardinality and N values inserted is:

(2 * log_2(L) * N) / 8 Bytes

+

L * 8 Bytes

2) I.e. :

the cost of a OZBC bitmap index on a int16 column (L = 2^16) with N values inserted is:

(2 * log_2(2^16) * N) / 8 + L * 8 =

2 * ((2 * log_2(2^8) * N) / 8) + L * 8

TWICE THE SIZE OF A OZBC BITMAP INDEX ON A int8 COLUMN

Validation

To test the performance in inserting and query time we have indexed a data set of 81,627,716 real IPv4 packets (provided by CAIDA) and we have created OZBC Bitmap Indexes on srcIP, dstIP, srcPort, dstPort:

- To index int16 columns (srcPort and dstPort) we use 2^16 OZBC Bitmap (1 bitmap for every distinct value).

- To index int32 columns (srcIP and dstIP) we decompose each int32 columns in two int16 and we used two indexes on int16, then we use 2^17 OZBC bitmap to index a int32 column. In this case we have 2 bits set for each row and we must read 2 bitmap for each equality query.

Index implementation : 1

In a bitmap index we need to have the whole index (composed from L bitmap) in memory at the insertion time. For these reasons we don’t have a single chunk of L bitmap that indexes all records inserted and we have decomposed each index in chunks. Each chunk (composed from L bitmap) indexes K records. The chunks are serialized one after the other and for each chunk we serialized the offset of all bitmap, so at the query time we can extract only the bitmap that are required.

We have created indexes with K = 30,000,000.

Index implementation : 2

In a query time we extract for each chunk 1 bitmap on index16 and 2 bitmap on index32 (in this case we need also compute a bitwise AND of the 2 bitmap extracted). The number of rows of each chunks is fixed and then we can make logical AND of 2 Index without decompress all bitmap.

Column implementation

We confront the query time of OZBC Bitmap Index vs a full scan on all records inserted. For optimized the query time without index if there are more then one conditions we take all ID_1 records that match with the condition_1 with a full scan and we take all ID_2 records that match with the condition_2 matching only ID_1 values and so on to the end condition.

Performance Tests

We have made performance test on:

1) The size of the index vs the size of data.
2) The time to create and write index.
3) The responsive query time.

Tests 2 and 3 are validate on two different PC:

PC_1:
- Intel Core i7-7500U – 2.70GHz – 4Mb Cache,
- Toshiba SSD - M.2 SATA 6.0G

PC_2:
- Intel Core I7-6700 – 3,4GHz – 8Mb Cache,
- Seagate Barracuda – Serial Ata3 – 64Mb7200.14

Index size vs Column size

The table shows the size of the indexes and columns on 81,627,716 records. We have created a column and a index on srcIP, dstIP, srcPort, dstPort.

	srcIP	dstIP	srcPort	dstPort	Total
Columns	311.6 MB	311.6 MB	155.9 MB	155.9 MB	935 MB
Indexes	367.2 MB	302.0 MB	195.4 MB	95.6 MB	960 MB

Insertion time

The table shows the time to create srcIP, dstIP, srcPort, dstPort indexes on 81,627,716 records.

	Total time	Insert / seconds	MB / seconds
PC_1 (ssd)	10.47 s	7,793,402	91 MB/s
PC_2 (no_ssd)	11.76 s	6,943,329	81 MB/s

Query time : 1

The table shows the time on 81,627,716 records to get all ID that satisfy the conditions:

- Q1 : srcIP = 137.195.64.142;

- Numbers ID founds = 778;

	no_Index	Index	no_Index / Index	Index / (no_Index / 100)
PC_1 (ssd)	688 ms	8 ms	87 x	1.2 %
PC_2 (no_ssd)	2,757 ms	192 ms	14.3 x	7 %

Query time : 2

The table shows the time on 81,627,716 records to get all ID that satisfy the conditions:

- Q2 : (srcIP = 145.220.214.145) AND (dstIP = 102.3.199.173);

- Numbers ID founds = 98,624;

	no_Index	Index	no_Index / Index	Index / (no_Index / 100)
PC_1 (ssd)	1,810 ms	31 ms	58 x	1.7 %
PC_2 (no_ssd)	3,674 ms	307 ms	12 x	8.3 %

Query time : 3

The table shows the time on 81,627,716 records to get all ID that satisfy the conditions:

- Q3 : srcPort = 53798

- Numbers ID founds = 22,438;

	no_Index	Index	no_Index / Index	Index / (no_Index / 100)
PC_1 (ssd)	341 ms	5.6 ms	60 x	1.7 %
PC_2 (no_ssd)	1,099 ms	82 ms	13.4 x	7.5 %

Query time : 4

The table shows the time on 81,627,716 records to get all ID that satisfy the conditions:

- Q4 : dstPort = 80

- Numbers ID founds = 24,796,963;

	no_Index	Index	no_Index / Index	Index / (no_Index / 100)
PC_1 (ssd)	386 ms	427 ms	0.92 x	108 %
PC_2 (no_ssd)	1,155 ms	534 ms	2.1 x	47.6 %

Query time : 5

The table shows the time on 81,627,716 records to get all ID that satisfy the conditions:

- Q5 : (srcIP = 145.220.214.145) AND
  (dstIP = 102.3.199.173) AND (srcPort = 80);
- Numbers ID founds = 98,624;

	no_Index	Index	no_Index / Index	Index / (no_Index / 100)
PC_1 (ssd)	2,356 ms	80 ms	29.4 x	3.4 %
PC_2 (no_ssd)	4,919 ms	385 ms	12.8 x	7.8 %

Conclusions : 1

Thanks to OZBC bitmap compression we have realized indexes on integers that can index millions of records per seconds and with a limited persistent storage size.

	Indexes size / Columns size	(create(Indexes) + write(Indexes)) / max_speed_write
PC_1 (ssd)	1.03	0.41
PC_2 (no_ssd)		0.51

The table shows the Indexes size compared to Columns size and compares the speed of creating and writing indexes with the maximum write speed of the storage device (calculated with “dd” command).

Conclusions : 2

The results show that OZBC Bitmap Index can improve significantly the query performance compared to full scan.

	Q1	Q2	Q3	Q4	Q5
ID founds	778	98,624	22,438	24,796,963	92,624
PC_1 (ssd)	87 x	58 x	60 x	0.92 x	29.4 x
PC_2 (no_ssd)	14.3 x	12 x	13.4 x	2.1 x	12.8 x

In the table is showed how fast are the indexes compared to query time without indexes. We can see that there is a one case (Q4 PC_2) that the Index query time it's worse then full scan query time. In this case the indexes don't improve query performance because the OZBC Bitmap size and then the time to decompress a OZBC Bitmap is proportional to the number of 1 bits and in this case there are 24,796,964 set bits.

Future Works

- We have seen that the Index query time depends by the number of 1 bits set of the bitmap extracted. To further decrease Index query time in this case, we are working to improve the OZBCBitmap unrolling implementation.

- In the actual Index implementation we need to make 2 “seek” operation for each Index chunk (one to extracts the bitmap offset and one to extracts the compressed bitmap). We are working on a new serialization scheme that   halved the number of the “seek” operation.