OPTIMIZED SEARCHING TECHNIQUE IN ROUTING

B.G.Prasanthi[1], Dr.T.Bhaskara Reddy[2.]

[1] Department of Computer Science, S.K.University,Anantapur
nitai2009@gmail.com
[2] Department of Computer Science, S.K.University,Anantapur
bhaskareddy_sku@yahoo.co.in

ABSTRACT

This algorithm describes technique to generate and manage unique integer indexes from a specified range of integers. Generated index can be used in any application where a unique integer from a specified range need to be served as key to a particular record and when record is freed index need to be reused. Efficiency of this algorithm lies in its simplicity and capability to manage large range of index in minimal recourses, such as running time and memory requirement.

This Algorithm is best suited for problems where 0 or 1-based unique indexes (however non 0 or 1 based indexes can also be managed with calculating a fix offset) are required to be managed with frequent operation like checking whether a index is free or not, finding first free index, reserving and freeing indexes with optimal memory usage in average case. One of the application but not limited to, is index generation for MIB tables where a unique index need to be used with a conceptual row for creation/retrieval/destroy operations.

KEYWORDS

Index generation, MIB, Chunk , Arbitrary Index, Reserve index, Bit state memory.

1. ALGORITHM

Algorithm works on bit state (0 or 1), one of the states is used to indicate free or used index in the memory. Thus 1 bit memory is required to represent one integer index, and state of this bit can be used to determine whether index is free or occupied.

For example to generate 512 integer indexes we need 512 bit memory that is 64 Byte. In addition to this we also need few more byte to make searching of the indexes faster and efficient to use.

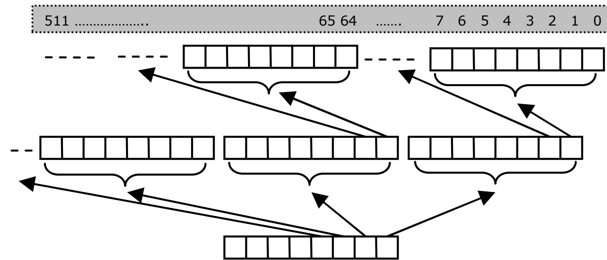

Figure 1. 8 bit base size

Above picture illustrate core of the algorithm for 8 bit base size. At level 0 has 8 bit memory which can point up to another 8 byte memory of level 1. Again at level 1 each bit can point to another 1 byte at next level. Thus level 1 has 8 byte memory and level 2 has 64 bytes of memory. If level 2 is our final level, then each bit at level 2 represents a unique index.

In above example level 2 has 64 Byte and is capable to manage 64*8 = 512 unique indexes.

Properties:

- If all bits are set in a chunk, bit pointing to this chunk at 1 level below must also be set to 1 if exist, else bit pointing to this chunk at 1 level below must be 0.
- Level 0 will have only one chunk.
- All other higher level can have max up to N chunk, where N is no. of total bits in immediate lower level.

1.1. Managing Indexes

Managing indexes are some important operations to manage indexes - Memory array at level MAX_LEVEL is used to generate and manage integer indexes. Considering bit state 0 for free indexes and 1 for occupied indexes. Here

1.2. Occupying an Index

Initially all bits at all levels will be set to 0. This indicates all indexes are free to be used.

```
reserveIndex (index)
h = MAX_LEVEL – 1
while h >= 0
p= index%BASE_SIZE
index = index/BASE_SIZE
idx_db[h][index] OR (1<<p)

if idx_db[h][index] = ALL_SET
h = h - 1
else
```

break

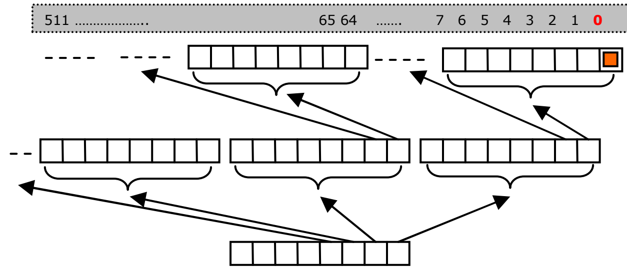

Figure 2. Reserve index 1

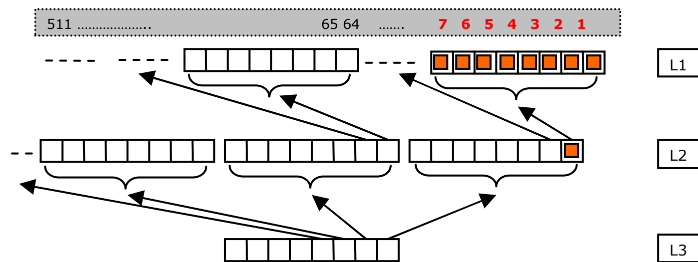

Figure 3. Reserve index 7, Case 2a

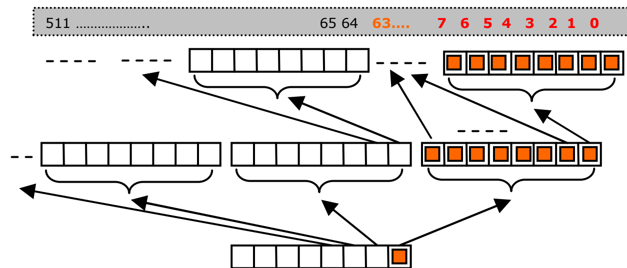

Figure 4. Reserve index 63, Case 2b

1.3. Releasing an Index:

Case1: When an index is released, corresponding bit at level 2 is set to 0 indicating that this index is now available to reuse.

If all bit on a single chunk (of size BASE_SIZE, in above example it is 8) are not occupied at current level, move one level below and set bit pointing to this chunk to 0.

Case1a: Repeat above step until, reached to level 0.

unreserveIndex (index)
h = MAX_LEVEL – 1
while h >= 0
p= index%BASE_SIZE
index = index/BASE_SIZE
if idx_db[h][index] = ALL_SET idx_db[h][index] AND ~(1<<p)
h = h - 1
else
idx_db[h][index] AND ~(1<<p)
break

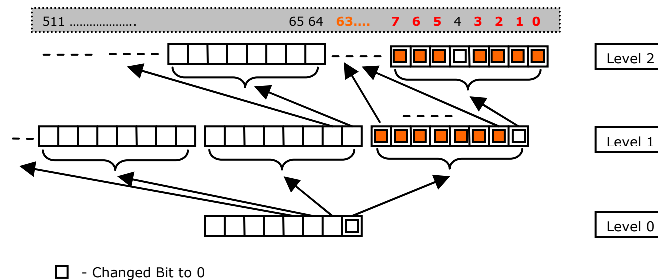

Fig.5: Releasing index 4, Case 1b

1.4. Searching First Free Index:

Let **p** be the position of first bit in the chunk which is 0 and **i** be the chunk no. of the current level which is being searched and **h** is current level.

This operation finds first free bit in a give chunk of size BASE_SIZE and returns its bit position from right.

findFree (h, i)
for **p** -> 0 to BASE_SIZE
if **idx_db[h][i]** & (1<<**p**) = 0
break
return **p**
This operation finds first smallest free index.

getFreeIndex (h , i)
while **h** < MAX_LEVEL
p = findFree(h, i)

if **p** >= BASE_SIZE
return -1

i = **i***BASE_SIZE + **p**
return **i**

1.5. Checking Index Status

This operation can determines whether a particular index is free or reserved. It has complexity O(1)
isIndexFree (index)
p= index%BASE_SIZE
i = **index**/BASE_SIZE
if **idx_db** [MAX_LEVEL-1]**[i]** & (1<<**p**) = TRUE
return FALSE
else
return TRUE

1.6. Memory Requirement

Optimal Memory Requirement for n indexes -

Total Memory in Bits = $\sum_{x=1,\, h} f(x)*b$

Where

n = Number of indexes
b = Size of base element
h = Ceiling ($\log_b n$)

$$f(x) = \begin{cases} \text{Ceiling } (n/b) & \text{for } x = h \\ & \end{cases}$$

Note: f(x) represents number of bits required at each level

For Example to generate 513 indexes with base element size 8 -

n= 513
b = 8

h = Ceiling ($\log_8 513$)

= Ceiling (3.0001)
= 4

f(4) = Ceiling(513/8)
= 65
f(3) = Ceiling(65/8)
= 9
f(2) = Ceiling(9/8)
= 2
f(1) = Ceiling(2/8)
= 1

Total Memory in Bits = $\sum_{x=1, 4} f(x)*8$
= 1*8 + 2*8 + 9*8 + 65*8
= 616 bits
~= 77 Bytes

With given h and b, Maximum b^h indexes can be managed. In this case memory requirement -

Total Memory in Bits = $\sum_{i=1, h} b^i$

Where

n = Number of indexes
= Size of base element
h = $\log_b n$

For Example for Above Case -
No of index to be generated (n) = 512
BASE_SIZE (b) = 8
MAX_LEVEL (h) = Ceiling ($\log_8 512$)
= 3

Total no of bits = $8^1 + 8^2 + 8^3$
= 8 + 64 + 512
~= 73 Bytes

Another example where we need to generate 32K indexes with chunk size 32.

No of index to be generated (n) = 32768
BASE_SIZE (b) = 32
MAX_LEVEL (h) = Ceiling ($\log_{32} 32768$)
= 3

Total no of bits = $32^1 + 32^2 + 32^3$
= 32 + 1024 + 32768
~= 4 Byte + 32 Byte + 4096 Byte
= 4132 Bytes

1.7. Running Time

This algorithm is extremely fast, number of index to be generated does not have much affect on running time of various operation.

At very broad level following is the no of operation that this algorithm performs for various cases –

1.7.1. Occupying an index:
Best Case – O (1)
Worst Case - O ($\log_b n$)

1.7.2. Releasing an Index:
Best Case – O (1)
Worst Case - O($\log_b n$)

1.7.3. Finding an Free Index: (Using sequential search for free bit in chunk)
Best Case – O ($\log_b n$)
Worst Case - O ($b.\log_b n$)

Table 1: Comparison table for managing 2^{15} indexes with different techniques:

	This Technique	Bit list	Link list	Range List
Memory	4132 Bytes	4096 Bytes	0 – 262144 Bytes	12 – 196608 bytes
Finding First Free Index	3 to 96 Comparison + d	1 to 1024 Comparison +d	1	1
Checking arbitrary Index Status	D	D	1 to 2^{15} Traversing and Comparison	1 to 2^{14} Traversing and Comparison
Reserving arbitrary Index	d to 3d	D	1 (need manage duplicates)	1 to 2^{14} + x + y
Freeing arbitrary Index	d to 3d	D	1 (need manage duplicates)	1 to 2^{14} + x + y
Remark			What if we need least free index? For this link list need to managed in sorted order	

d = delay to calculate offset and index + set/reset/check bit status
x = delay to merge/split range list.
Y = delay to allocate and insert a node

2.0. CONCLUSION

When compared to the bit list, link list and range list this is efficient & lies in its simplicity and capability to manage large range of index in minimal recourses, such as running time and memory requirement.

This Algorithm is best suited for problems where 0 or 1-based unique indexes (however non 0 or 1 based indexes can also be managed with calculating a fix offset) are required to be managed with frequent operation like checking whether a index is free or not, finding first free index, reserving and freeing indexes with optimal memory usage in average case.

1.8. References

[1] Reynold cheng,Yuny Xia and Rahul shah ,"Efficient indexing methods for probabilistic threshold queries over uncertain data",IN-47907-1392,USA

[2] R. Cheng, D. V. Kalashnikov, and S. Prabhakar (2004). Querying imprecise data in moving object environments. IEEE Transactions on Knowledge and Data Engineering (To appear),

[3] R. Cheng and S. Prabhakar (2008). Managing uncertainty in sensor databases. In SIGMOD Record issue on Sensor Technology.

[4] Kenneth L. Clarkson (2007). New applications of random sampling in computational geometry. Discrete and Computational Geometry, 2:195{222}

[5] Je_ Erickson and Pankaj K. Agarwal (2007). Geometric range searching and its relatives. Advances in Discrete and Computational Geometry, Contemporary Mathematics 223:1{56}

[6] Michael L. Fredman (2006). Lower bounds on the complexity of some optimal data structures. SIAM J. Comput., 10(1):1{10}

[7] Jonathan Goldstein, Raghu Ramakrishnan, UriShaft, and Jie-Bing Yu(1997) Processing queries by linear constraints. In PODS, pages 257{267}

[8] Paris C. Kanellakis, Sridhar Ramaswamy, Darren Erik Vengro_, and Je_rey Scott Vitter (1996). Indexing for data models with constraints and classes. J. Comput. Syst. Sci, 52(3):589{612}

[9] H. Kriegel, M. Potke, and T. Seidl (2006). Managing intervals e_ciently in object-relational databases. In Proc. of the 26th Intl. Conf. on VLDB, Cairo, Egypt.

[10] B. Lin, H. Mokhtar, R. Pelaez-Aguilera, and J. Su (2003). Querying moving objects with uncertainty. In Proceedings of IEEE Semiannual Ve- hicular Technology Conference.