The International Journal Of Engineering And Science (IJES)
||Volume||2 ||Issue|| 8 ||Pages|| 41-49 ||2013||
ISSN(e): 2319 – 1813 ISSN(p): 2319 – 1805

Array Indexing Technique: Answering The Iceberg Queries Efficiently

Ankam Praveen, Vuppu Shankar
Department of Computer Science and Engineering, KITS, Warangal. Andhra Pradesh, India.
Associate Professor
Department of Computer Science and Engineering, KITS, Warangal Andhra Pradesh, India.

----------ABSTRACT----------
The effectiveness of a data retrieving method depends upon the data specific queries for retrieving the data from the database. Fundamentally, iceberg queries are unique class of aggregation queries that compute aggregate values upon user interested threshold. The basic bitmap index technique is used to process an iceberg query by conducting a bitwise-AND operation between every pair of bitmap vectors that consumes a large amount of execution time to answer an iceberg query. Furthermore, this execution time increases when the cardinality of an attributes are increases. The major part taken into the consideration about the bitwise-AND operations in the iceberg queries. The reduced number of AND operation increases the effectiveness of the iceberg query. In this work, an efficient iceberg query evaluation process is proposed by removing the unnecessary fruitless bitwise-AND operations needed to find the item pairs. The proposed scheme is index-based technique. In this scheme, the index positions of bitmaps vectors whose 1's are retrieved. Then retrieved index positions will be further processed to turn out in to iceberg result upon user provided threshold. The extensive experimental results on the real and synthetic data sets are showing better execution time of an iceberg query evaluation.

KEYWORDS: *Bitmap index, Bitwise-AND operation, Bitmap vectors, Database, Information retrieval system.*

Date of Submission: Date 12Aug. 2013, Date of Publication: 30 Aug2013,

I. INTRODUCTION

Discovery of knowledge or summarized information from operational databases which is frequently required by the top officials and/or executives to make better decisions in modern business organizations. Then an aggregation is one kind of knowledge representation which is computed by processing of one or more selected attributes of the database table and is useful to perform business analysis by computer analyst.

Iceberg queries were first studied in database and data mining field by scientist named M. Fang et.al [12]. According to him, IBQ is defined as this is a unique class of an aggregation query which computes aggregated values upon user provided thresholds. Syntax of an IBQ on a relational table R (C_1, C_2… C_n) is stated below:

SELECT C_i, C_j… C_m, AGG (*) FROM R GROUP BY C_i, C_j… C_m HAVING AGG (*) > = T

Where **C_i, C_j… C_m** represents a subset of attributes in R and referred to as aggregate attributes. AGG represents an aggregation function such as COUNT, SUM, MIN and MAX. The greater than or equal to (>=) is a special symbol used as a comparison predicate.

In this paper we focus on iceberg query with aggregation function COUNT having the anti-monotone property. For example, if the count of a group is below T the count of any super group must be below T. Iceberg queries are today being processed with techniques that do not scale well to large data sets. Hence, it is necessary to develop efficient techniques to process them easily. One simple technique to answer an iceberg query is by first aggregating all tuples using GROUP BY clause and then evaluating the HAVING clause to select the qualified tuples among them. However, this is difficult because database table is several times larger than main memory. In another technique, the records of the database table were sorted on the disk and then passed the sorted records into the main memory to form an aggregation. Further, it selects values of an aggregation which are greater than a specified threshold. If the available memory is less than the table size, then the data is to be passed over in more number of times from the disk.

Therefore, query evaluation consumes long execution time. It is observed that, above techniques evaluate iceberg queries by offering a long CPU time and sufficient main memory, but in practice these two are costlier resources in the computer systems.

Bin He et al. [1], evaluated iceberg query quickly by minimizing the CPU and Memory usage statistics using compressed bitmap index. He indexed all the bitmap vectors of attributes in the selection. A bitmap for an attribute in a table can be viewed as a matrix having r rows consisting corresponding number of tuples and columns indicating the number of distinct values of an attribute. If there is a bitmap vector in the k^{th} position of the attribute then the element in the matrix is 1 else 0. Then the original bitmap vectors were aligned with available free space in the memory using word aligned hybrid compression technique. Bin He et.al [1] used priority queues to efficiently evaluate the iceberg queries. The bitmap vectors were placed in the said priority queues an on order of their first 1 bit position computed by function First1bitposition. Then bitwise- AND operation was conducted on the vectors pair that were identified by other function called FindNextAlignedVector. In order to select the next aligned vectors pair, the push and pop operations performed repeatedly until either of the PQs becomes empty.

In this, we consider iceberg query with count function and allows anti-monotone property which is stated as "if count of any group is below threshold then its super group must be below threshold ".The iceberg query is evaluated by conducting a bitwise-AND operation between every pair of bitmap vector. If the resultant vector of that pair have enough 1's count upon user provided threshold then declared as iceberg result, if not, not an iceberg result. Else this is also verified for empty or zero vectors. That means the vector which does not contain any 1's in it. Thus the AND operation between them is wasted except in first case which is being an iceberg result. Further, this process was also complicated to increased number of bitmap pairs that are increased by the cardinality of attributes are increased.

However, the empty bitwise-AND result problem was completely solved recently by an author Bin He et.al in [1] by developing an efficient vector alignment algorithm. The vector alignment algorithm ensures that no empty bitwise-AND results will be generated before conducting any AND operation. In this efficient work, the IBQ execution time is improved greatly by eliminating AND operation between pair of bitmap vector whose resultant is empty or zero. But in this case, the query evaluation time increased is for every bitwise-AND operation on 2 vectors takes place, for that we need to scan each pair when there is a vector alignment of 2 vectors is found otherwise the search time will be major factor in finding the item pair of iceberg query.

For Example: If A1 is a vector of an attribute A and B1, B2 and B3 are vectors of B attribute then we need to find the alignments for A1 with B1, A1 with B2 and A1 with B3. In this case A1 should scan for 3 times with B vector to find vector alignment. Thus causes wasting of search time for finding the same for vector alignment with B then performing AND operation on them.

In this paper, a new scheme of evaluation of iceberg query is proposed to avoid AND operations between pair of bitmap vectors. The proposed system improves further simplification of evaluation process using index-based technique. In this scheme, the index positions of all bitmap vectors of each attribute, whose index positions 1's are retrieved and that are further processed to turn out into iceberg results upon user provided threshold. With this approach, we do not perform bitwise-AND operation and we can declare an iceberg item pairs without performing any bitwise-AND operation. The index-based accessing of the positions of bitmap vectors will increases the execution time.

II. LITERATURE SURVEY

In recent times, the evaluation of iceberg queries has attracted researchers significantly due to the demand of scalability and efficiency. The research work is reviewed in two subsections. In the first subsection i.e. 2.1, we provide a review of related work on the bitmap index technology used by different authors. We also review the related research work of an iceberg query evaluation using bitmap indices in second subsection i.e. 2.2 which is the focus of this paper for optimization of iceberg queries.

2.1 Bitmap index

The concept of bitmap index was first introduced by professor Israel Spiegler et al [19]. Bitmap indices are known to be efficient in order to accelerate the iceberg queries especially used in the data warehousing applications and in column stores. In data warehouse applications, bitmap indices are shown to perform better than tree based index scheme, such as the variants of B-tree or R-tree [13], [15], [17].

Compressed bitmap indices are widely used in column oriented data bases, such as C-store [14] to improve the performance over row oriented data bases.

Table 1: An Example of Bitmap Index

A	B
A1	B1
A2	B2
A3	B1
A2	B1
A1	B3
A2	B2
A1	B2
A1	B2
A3	B1

(a) Table R

A1	A2	A3
1	0	0
0	1	0
0	0	1
0	1	0
1	0	0
0	1	0
1	0	0
1	0	0
0	0	1

B1	B2	B3
1	0	0
0	1	0
1	0	0
1	0	0
0	0	1
0	1	0
0	1	0
0	1	0
1	0	0

(b) Bitmap Indices for A & B

2.2 Iceberg query evaluation

Processing of iceberg query was first studied by Fang et.al [12] by extending the probabilistic techniques [11] and suggested hybrid and multi buckets algorithms. The sampling and multiple hash function techniques were used as basic building blocks of probabilistic techniques such as scaled-sampling and coarse-count algorithms.

They estimated the sizes of query results in order to predict the valid iceberg results. This improves query performance and reduces memory requirements greatly. However, these techniques erroneously resulted in false positives and false negatives. To recover from these errors, efficient strategies are designed by hybridizing the sampling and coarse-count techniques. To optimize the query execution time of hybrid strategies by extending the linear counting probabilistic algorithm for counting the number of unique values in the presence of duplicates. The linear counting algorithm is based on hashing technique allocates a bitmap (hash table) of size m in main memory. All entries are initialized to "0"s. The algorithm then scans the relation and applies a hash function to each data value in the column of interest. The hash function generates a bitmap address and the algorithm sets this addressed bit to "1". The algorithm first counts the number of empty bitmap entries. It then estimates the column cardinality by dividing this count by the bitmap size m and plugging the result.

2.3 Existing System

In the existing approach, the bitwise-AND operations were performed on each pair of bitmap values of Vector A_i and B_j when first aligned positions were found. So with that approach, there is a chance of performing unsuccessful AND operations on 2 vectors. Thus, causes the vain of bitwise-AND operations and leads to increase the execution time. The following picture depicts the bitmaps vector A and vector B before performing AND operation.

Table2: Bitmap Vectors of A1 and B1

Vector/ Index Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1	0
B1	1	0	0	0	1	1	0	0	1	0	1	1	1	0	0	0	0

From the Table 2. **The first aligned position is** found **at 4** for vector A1 and vector B1. So the bitwise-AND operation takes place from this position and operation will continued to till the end of the two vectors. The following picture depicts **after bitwise-AND operations the resultant vector** will be:

Table3: Bitwise-AND operation between Bitmap Vectors of A1 and B1

Vector/ Index Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1	0
B1	1	0	0	0	1	1	0	0	1	0	1	1	1	0	0	0	0
Result					1	1	0	0	0	0	1	1	0	0	0	0	0

From the Table 3. The **red shaded cells** representing **unsuccessful AND operation results**. The length of vector A1 and vector B1 is 17. So, total 9 bitwise-AND operations are performing unnecessarily after first aligned position found at 4, even the results are zero. We aimed to overcome this kind of problem with our proposed work.

III. RESEARCH ELABORATION

The aggregated attributes mentioned in the iceberg query are read from the database table and generate equivalent bitmaps of them. Then index positions of 1's of bitmaps A_m and B_m are retrieved. The retrieved position values of A_m and B_m are stored into an array called aIndex and bIndex respectively. Then Vector A_m index positions are compared with vector B_m index positions. If they are equal, we maintain a counter that will be incremented by 1 every time when there is a common position of 1's found and then compare the counter value with iceberg threshold, if it is above threshold, then confirm this vector pair as an iceberg result and include them into iceberg result set.

Then the resultant index positions of result vector are made it to zero with original vectors A_m and B_m index positions. The updated bitmap vectors of A_m and B_m to compare bit index positions with another bitmap vector in the next iteration. Otherwise, we prune the bitmap A_m and B_m if index count not passes the threshold. Continue the same process until all the vector pairs are completed.

Example bitmap vectors of A1 and B1:

Vector/ Index Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1	0
B1	1	0	0	0	1	1	0	0	1	0	1	1	1	0	0	0	0

Fig 1. Shows that the retrieving of all 1-bit index positions of A1 into **aIndex** array and retrieving all 1-bit index positions of B1 into **bIndex** array.

We explain the proposed work in the following steps:

Step1: **Retrieve the Index-positions**

Here aIndex is an array which stores the index-positions of all 1-bit positions of A1 vector. And we represent First 1-bit position of A1 can be pointed by fp(first position) and last 1-bit position A1 can be pointed by lp (last position). Similarly we do the same for B1 vector. aIndex array consists of all 1-bit position index values of A1. And bIndex array consists all 1-bit position index values of B1.

Step2: **Check the Threshold with aIndex and bIndex**

After retrieving the index-positions of vector A1 and B1, we check the aIndex length and bIndex length with the Threshold T. If the length of the aIndex array passes the threshold value T (i.e., Length of aIndex >= T) then the vector A1 is eligible for processing, if not we simply ignore the vector A1 and we go for other vector of Attribute A(i.e. A2, A3, A4 and so on). We do the same operation for vector B. (i.e. Length of bIndex >= T) then the vector B1 is eligible for processing if not ignore the vector B1 and select the other vector of B for processing.

Step3: **Set the boundaries of fp and lp to compare 2 vectors**

After the vector A1 and B1 passes the threshold value then, the process will start now for comparison. Here we do compare aIndex and bIndex arrays and we maintain a counter that counts the occurrences of aIndex values found in bIndex array.

First we check the **fp of aIndex** with **fp of bIndex, if** aIndex.fp > bIndex.fp **then** we perform the comparison between aIndexFp & bIndexFp to aIndexLP & bIndexLP **else** we do comparison operation between bIndexFP & aIndexFP to bIndexLP & aIndexLP. This will fix the boundaries for comparison operations of aIndex and bIndex.

Step4: **Compare the aIndex and bIndex**

By setting the FP and LP that limits the comparison operation that takes place in between the range only. Now, compare aIndex array value with bIndex array value. In this process of comparison we maintain a counter that counts the occurrences.

If aIndex array value found in bIndex array then we increment the counter by 1. The process of comparison will continues to till the end of the aIndex and bIndex .If the occurrence of aIndex found in bIndex then we store the value that is occurred into the resIndex (result Index array)

Step5: Check the Threshold with resIndex

After all comparisons of aIndex and bIndex arrays, we check the length of resIndex array where it stores the occurred value., i.e. we check resIndexLength > T ,then **we declare the pair A1 and B1 as Iceberg result,** because for first time A1 and B1 vectors index values stored inside aIndex and bIndex.
For example, consider the user provided threshold value **T = 4** then,

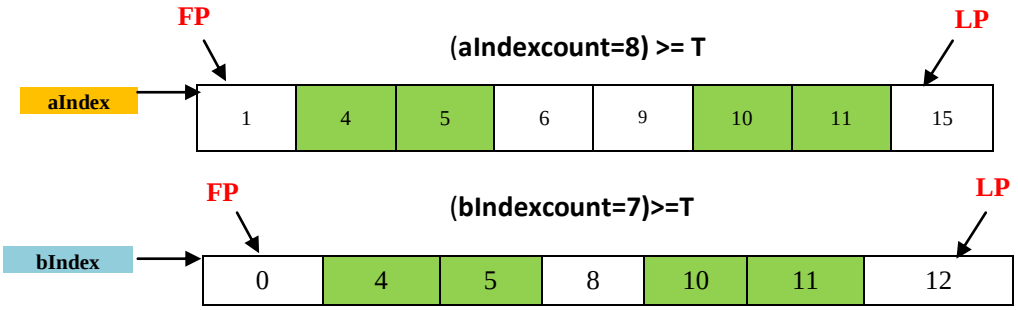
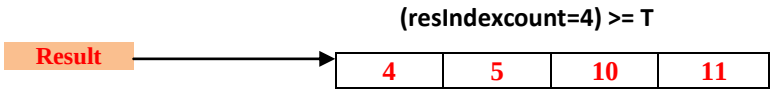


Fig1: Retrieved Index-positions into an arrays aIndex and bIndex

Result array count passes the threshold, so declaring A1 and B1 pair as iceberg result.



The Fig.2 depicts, after Subtract **operation** of **result array** with **aIndex** and **bindex** will be updated:

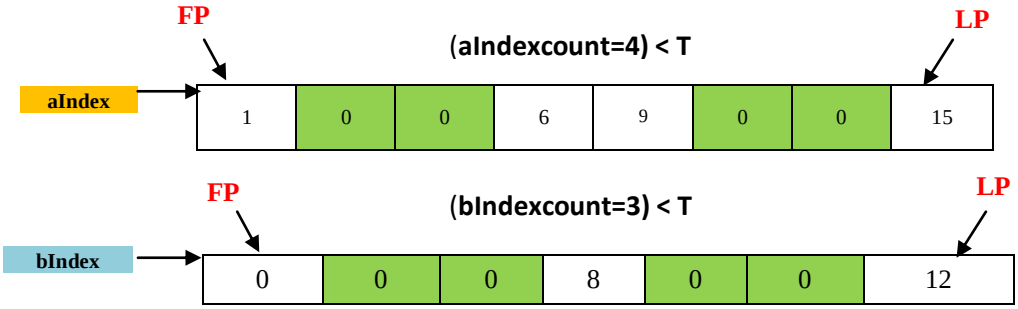


Fig2: Retrieved Index-positions into an arrays aIndex and bIndex

We continue the same process for A1 with other B_m because A1 passes the threshold value, and ignore B1 vector for next iteration with A_m because it does not passes threshold value.

3.1 Implementation

The following algorithms will explain that how we implemented the proposed work. Here we used the algorithms named 1. **IcebergQueryEvaluation** **2. FindAllOnePositions** **3. PerformSubtract**

In this section we explained the algorithms that will be useful for implementing.

Proposed algorithms to evaluate iceberg query using indexing positions.

Algorithm 1: IcebergQueryEvaluation
IcebergQuery (Attribute A, Attribute B, threshold T)
Output: Iceberg Results

```
1.   For each bitmap vector a of Attribute A do     // this loop will finds all 1-bit index positions of vector a&b
2.              aIndex[ ],  a.countOnes, a.fp,a.lp=findAllOnePositions(a)
3.   //    fp is the first 1-bit position, lp is the last 1-bit position
4.   For each bitmap vector b of attribute B do
5.              bIndex[ ],b.countOnes   b.fp,b.lp=findAllOnePositions(b)
6.    While a ≠ null and b ≠ null do  // Finding all common index positions of a&b
7.                  countres=0; k=0
8.       If  ( a.countOnes >= threshold && b.countOnes >= threshold) then
9.                If( a.fp >= b.fp and a.lp>= b.lp) then
10.                For i=a.fp to a.lp step 1 do
11.                   If (aIndex[i] = bIndex[i]) then
12.                                Res[k]=i;
13.                                 Increment k by 1.              // k = k + 1.
14.                                 Increment countRes by 1     // countres=countres+1
15.                    End if
16.          Else
17.             For i=b.fp to b.lp step 1 do
18.                   If(aIndex[i] = bIndex[i]) then
19.                                Res[k]=i;
20.                                 Increment k by 1.            // k=k+1
21.                                Increment countres  by 1   // countres=countres+1
22.                    End if
23.          End if
24.       If ( countres >= threshold ) then                  //ibq evaluation
25.                  Add Iceberg result(a.value, b.value, countres) into R
26.       a,b=performXOR(aIndex,bIndex,Res)
27.         a.countOnes=getCount(aIndex, a.fp,a.lp)
28.         b.countOnes=getCount(bIndex,b.fp,b.lp)
29.     If(a.countOnes > threshold ) then
30.        Repeat from step 9 to 30 for  a and b.next
31.   Else
32.        Repeat from step 9 to 30 for  a.next and b
33.   Return R
```

The above algorithm is efficiently designed to evaluate iceberg query as per proposed research work in the previous section of this paper. The algorithm is divided in to three phases. In the first phase i.e. from the line numbers from 1 to 4, the algorithm finds all index positions of 1's in each bitmap vector and returns into new array called aIndex[ ] with those index positions and it also computes count of index positions. In the second phase i.e. line numbers from 5 to 22,

---

Algorithm 2: FindAllOneBitPositions
findAllOnePositions (attribute X)
Output: finding fp, lp, returning xIndex []

```
1.        k=0,lp=0,fp=0,countone
2.        For i=0 to x.length step 1
3.             If x[i]=1 then
4.                xIndex[k]=i
5.               countone=count+1
6.                lp=i
7.                k=k+1
8.           Next i
9.        fp=xIndex[0]
10.       Return lp,fp,xIndex[],countone
```

The above algorithm which finds all 1-bit positions of Vector X. In this process, From line number 2 to 8, that will scan each bit, if it found a bit 1 then it stores the corresponding index value into and array called

xIndex[], and after scanning total records it will returns the last position of 1-bit found and first 1-bit position and also returns the array which contains the all index positions of 1's.

Algorithm 3: getCountofOnes
getCount (yIndex [], firstPosition, lastPosition)
Output: Count the number of positions having 1's in the vector.

```
1.    count=0;
2.    For i=firstPosition to lastPosition step 1
3.          If(yIndex[i]!=0) Then
4.              count=count+1
5.          End if
6.    Next i
7.    Return count
```

The above algorithm which finds the total number of index positions which were stored in the xIndex[] array. The algorithm will repeat the process for finding the 1's located in the vector.

Algorithm 4: performing Subtract operation
performSubtract (aaIndex, bbIndex, Res)
Output: returning updated aIndex, bbIndex.

```
1.    For i=0 to aaIndex.length step 1
2.        If (aaIndex[i]=Res[i]) then
3.            aaIndex[i]=0
4.        End if
5.    Next i
6.    For i=0 to bbIndex.length step 1
7.        If (bbIndex[i]=Res[i]) then
8.            bbIndex[i]=0
9.        End if
10.   Next i
11.   Return update aaIndex, bbIndex
```

The above algorithm which updates the aIndex array values by comparing the Res[] index array with aaIndex and bbIndex array. If aaIndex array value and Res[] array value equals then it changes the aaIndex [] array value to zero, and same will be applied to bbIndex[] array.

IV. RESULT & DISCUSSION

Our experiments were carried out with both a synthetic data set and a real patent data set. We generate large synthetic data according to five parameters: data size (i.e., number of tuples), attribute value distributions, number of distinct values, number of attributes in the table, and attribute lengths. In each experiment, we usually focus on the impact of one parameter with respect to data sizes and thus fix the other parameters.

4.1 Experimental set up

The experiments are conducted on a following computer system with Core2 Duo. Processor of 2.0 GHz, 1.0GB main memory running on Microsoft Windows XP ver., and all algorithms are implemented in java platform. The experiments are carried out repeatedly a minimum of 3 to 4 times of same threshold with synthetic data set consists of 1 lakh records. The experiments are conducted for high threshold values such as from 1000 to 10000.

4.2 Experimentation

In our experimentation, we observed that when the threshold value increased, the execution time decreased. We also observed that the array indexing technique gives faster results than the existing approaches. The following picture shows the experimentation result that was observed from IBQ using bitmap index technique and array indexing technique:

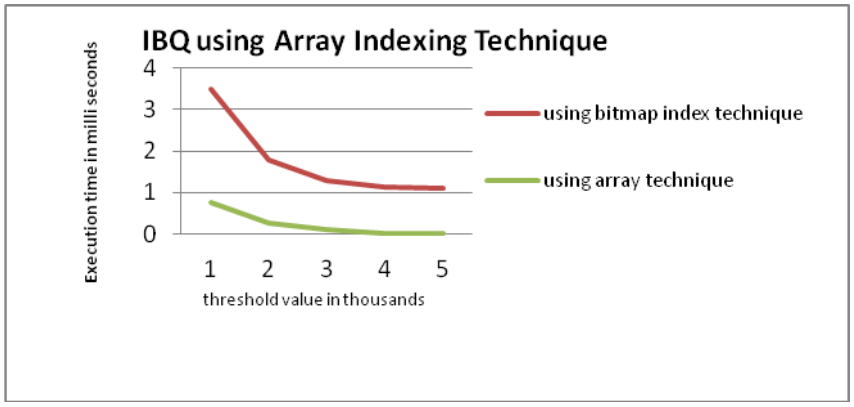


Fig 3: Execution times in icebergBitmap Index technique, icebergArrayIndexTechnique varying thresholds.

4.3 Comparative Analysis

The existing method refers to the efficient iceberg query evaluation using compressed bitmap index by Bin He *et al*. [1]. The comparison study is based on the responses of retail dataset with two different algorithms. Both the algorithms are based on the bitmap index table and the AND operation between the attributes. The aim of the algorithms is to reduce the execution time by reducing the unwanted AND operation. The fig.3 shows the comparative analysis of the proposed iceberg evaluation algorithm and the existing iceberg evaluation algorithm. Thus we can state that the proposed iceberg evaluation has quick response than the existing approach mentioned in [1].

V. CONCLUSION

In the field of information retrieval, the data retrieval from the database is become more time consuming process as the number of data is increasing day by day. Specialized queries are used for retrieving data from the databases. Iceberg query are similar queries which uses aggregate function and conditional clauses to get the data from the databases quickly. The proposed iceberg query evaluation algorithm uses bitmap index table for the iceberg evaluation. The algorithm has two important things, one is declaring a vector pair as Iceberg Result without performing any AND operation. The second is to improve the efficiency of AND operations by Indexing approach for accessing elements faster. In another direction, the execution time can also be reduces by eliminating large number of unproductive bitwise-AND operations is would be taken up. Our algorithm demonstrates superior performance over existing schemes and it does not depend on any particular method. To solve the problem of massive empty and fruitless AND results, we proposed an efficient Indexing approach that gives Iceberg results without performing any bitwise-AND operations.

REFERENCES

[1] Bin He, Hui-I Hsiao, Ziyang Liu, Yu Huang and Yi Chen, "Efficient Iceberg Query Evaluation Using
[2] Compressed Bitmap Index", IEEE Transactions On Knowledge and Data Engineering, vol 24, issue 9,
[3] Sept 2011, pp.1570-1589.
[4] D.E. Knuth, "The Art of Computer Programming: A Foundation for computer mathematics"
[5] Addison- Wesley Professional, second edition, ISBN NO: 0- 201-89684-2, January 10, 1973.
[6] G.Antoshenkov, "Byte-aligned Bitmap Compression", Proceedings of the Conference on
[7] Data Compression, IEEE Computer Society, Washington, DC, USA, Mar28-30,1995, pp.476
[8] Hsiao H, Liu Z, Huang Y, Chen Y, "Efficient Iceberg Query Evaluation using Compressed
[9] Bitmap Index", in Knowledge and Data Engineering, IEEE, Issue: 99, 2011, pp:1.
[10] JinukBae, Sukho Lee, "Partitioning Algorithms for the Computation of Average Iceberg
[11] Queries", Springer-Verlag, ISBN:3-540-67980-4, 2000, pp: 276 – 286.
[12] J.Baeand, S.Lee, "Partitioning Algorithms for the Computation of Average Iceberg Queries", in
[13] DaWaK, 2000.
[14] K. P. Leela, P. M. Tolani, and J. R. Haritsa."On Incorporating Iceberg Queries in Query
[15] Processors", in DASFAA, 2004, pages 431–442.
[16] K.Stockinger, J.Cieslewicz, K.Wu, D.Rotem and A.Shoshani. "Using Bitmap Index for Joint
[17] Queries on Structured and Text Data", Annals of Information Systems, 2009, pp: 1–23.
[18] K.Wu, E.J.Otoo and A.Shoshani. "Optimizing Bitmap Indices with Efficient Compression",
[19] ACM Transactions on Database System, 31(1):1–38, 2006.
[20] K.Wu,EJ.Otoo,and A.Shoshani, "On the Performance of Bitmap Indices for High Cardinality
[21] Attributes", VLDB, 2004, pp: 24–35.
[22] K.-Y.Whang, B.T.V.Zanden and H.M.Taylor."A Linear-Time Probabilistic Counting Algorithm
[23] for Database Applications". ACMTrans.Database Syst., 15(2):208–229, 1990.
[24] M.Fang, N.Shivakumar, H.Garcia- Molina, R.Motwani and J.D.Ullman."Computing Iceberg
[25] Queries Efficiently". In VLDB, pages 299–310, 1998.

[26] M.Jrgens "Tree Based Indexes vs. Bitmap Indexes: A Performance Study" In DMDW, 1999.
[27] M.Stonebraker, D.J.Abadi, A.Batkin, X.Chen, M.Cherniack, M.Ferreira, E.Lau,A.Lin,
[28] S.Madden, E.J.O"Neil, P.E.O"Neil, A.Rasin, N.Tran and S.B.Zdonik.C-Store: "A Column-
[29] oriented DBMS". In VLDB, pages 553–564, 2005.
[30] P.E.O Neil."Model204 Architecture and Performance". In HPTS Pages 40–59, 1987.
[31] P.E.O Neiland D.Quass. "Improved Query Performance with Variant Indexes". In SIGMOD
[32] Conference, pages 38–49, 1997. 1182
[33] P.E.O Neil and G.Graefe. "Multi-Table Joins Through Bitmapped Join Indices". SIGMOD
[34] Record, 24(3):8–11, 1995.
[35] R.Agarwal, T.Imilinski, and A.Swami "Mining Association Rules between Sets of Items in
[36] Large databases". In SIGMOD Conference, pages 207-216, 1993.
[37] Spiegler I; Maayan R "Storage and retrieval considerations of binary databases". Information
[38] processing and management: an international journal 21 (3): pages 233-54, 1985
[39] Dr.C.V.GuruRao and V.Shankar, "Efficient iceberg query evaluation using compressed bitmap
[40] indices by deferring bitwise-XOR operations" 3rd IEEE, on Advanced Computing Conference,
[41] 22nd &23 rd Feb 2013,New Delhi, India,pp.1374-79.

Authors Profile:

Ankam Praveen obtained his Bachelor's degree in Computer Technology from KITS, Warangal, A.P., India. Pursuing his Master's degree in Software Engineering from KITS, Warangal, AP, India. He is currently working as an Assistant Professor at the Faculty of Computer Science and Engineering, Ganapathy Engineering College, Warangal. His specializations include Data mining and Data warehousing, Databases and Image Processing. His current research interest in Computation and evaluation of iceberg queries efficiently.

Vuppu Shankar obtained his Bachelor's degree in Computer Technology from Nagpur University of India. Then he obtained his Master's degree in Computer Science from JNTU Hyderabad and he is also life member of ISTE. He is currently an Associate Professor at the Faculty of Computer Science and Engineering, Kakatiya Institute of Technology & Science (KITS), Kakatiya University-Warangal. His specializations include Data mining and Data warehousing, Databases and networking. His current research interest in computation of an iceberg cubes and evaluation of iceberg queries efficiently.