On Incorporating Iceberg Queries in Query Processors

Krishna P. Leela[1], Pankaj M. Tolani[1], and Jayant R. Haritsa[1]

Dept. of Computer Science & Automation
Indian Institute of Science, Bangalore 560012, INDIA
{leekris,pankaj,haritsa}@csa.iisc.ernet.in

Abstract. Iceberg queries are a special case of SQL queries involving GROUP BY and HAVING clauses, wherein the answer set is small relative to the database size. We present here a performance framework and a detailed evaluation within this framework of the efficiency of various iceberg query processing techniques. Based on these results, we provide a simple recipe algorithm that can be implemented in a query optimizer to make appropriate algorithmic choices for processing iceberg queries.

Keywords: Iceberg Query, Query Optimizer

1 Introduction

Many database applications, ranging from decision support to information retrieval, involve SQL queries that compute aggregate functions over a set of grouped attributes and retain in the result only those groups whose aggregate values satisfy a simple comparison predicate with respect to a user-specified threshold. Consider, for example, the "Dean's Query" shown below for the Relation REGISTER (RollNo, CourseID, Credits, Grade):

```
SELECT RollNo, SUM(Credits)
FROM REGISTER
GROUP BY RollNo
HAVING SUM(Credits) > 18
```

This query returns the roll number of students currently registered for more than 18 course credits (i.e. the fast-track students). Here, the grouping attribute is the student roll number, the aggregate operator is SUM, the comparison predicate is "greater than", and the threshold value is 18. When the threshold is sufficiently restrictive such that the results form only a small fraction of the total number of groups in the database, the query is called an **iceberg query** [4] – the analogy is that the database is the iceberg and the small result represents the proverbial "tip" of the iceberg.

Database systems currently do not employ special techniques to process iceberg queries operating on large databases. That is, *independent* of the threshold value, they typically use the one of the following approaches:

Sort-Merge-Aggregate (SMA) : The relation is completely *sorted* on disk with regard to the group-by attributes and then, in a single sequential scan of the sorted database, those groups whose aggregate values meet the threshold requirement are output; or

Hybrid-Hash-Aggregate (HHA) : The relation is recursively *partitioned* using hash functions, resulting in partitions in which the distinct groups fit in the available main memory, where they are subsequently processed.

In general, these strategies appear wasteful since they do not take the threshold predicate into account, that is, they are not output sensitive. Motivated by this observation, a variety of customized algorithms for efficiently handling iceberg queries were proposed and evaluated in [4] by Fang et al. These algorithms, which we will collectively hereafter refer to as **CIQE**[1], are based on various combinations of sampling and hashing techniques. For example, the *Defer-Count* algorithm operates in the following manner: in the sampling scan, a random sample of the database is used to identify "candidate" (i.e. potentially qualifying) groups by scaling the sample results to the database size, followed by a hashing scan of the database to identify other candidate groups, winding up with a counting scan of the entire set of candidates against the database to identify exactly those that do meet the threshold requirement.

1.1 CIQE Applicability

`CIQE` represents the pioneering work in defining and tackling iceberg queries. However, it can be utilized only in a restricted set of iceberg query environments – specifically environments in which

1. The aggregate values of the groups have a highly skewed distribution; and
2. The aggregate operator is either COUNT or SUM; and
3. The comparison predicate is $>$.

An implication of the first constraint (high skew) is that `CIQE` would *not work* for the Dean's Query since the number of credits taken by students typically occupies a small range of values (in our institute, for example, the values range between 0 and 24, with 99 % of the students taking between 6 and 18 credits).

With respect to the second constraint, apart from COUNT and SUM, other common aggregate functions include MIN, MAX and AVERAGE. For example, an alternative "Dean's Query" could be to determine the honors students by identifying those who have scored better than a B grade in all of their courses. The candidate pruning techniques of `CIQE` are *not effective* for such aggregates since they introduce "false negatives" and post-processing to regain the false negatives can prove to be very expensive.

Finally, the impact of the third constraint ($>$ comparison predicate) is even more profound – restricting the predicate to $>$ means that only "High-Iceberg" queries, where we are looking for groups that *exceed* the threshold, can be supported. In practice, however, it is equally likely that the user may be interested in "Low-Iceberg" queries, that is, where the desired groups are those that are *below* a threshold. For example, an

[1] Representing the first letters of the paper's title words: Computing Iceberg Queries Efficiently.

alternative version of the "Dean's Query" could be to find the part-time students who are taking *less than* 6 credits.

At first sight, it may appear that Low-Iceberg queries are a simple variant of the High-Iceberg queries and can therefore be easily handled using a CIQE-style approach. But, in fact, the reality is that Low-Iceberg is a *much harder* problem since there are no known efficient techniques to identify the lowest frequencies in a distribution [8]. A practical implication is that the sampling and hashing scans that form the core of the CIQE algorithm fail to serve any purpose in the Low-Iceberg scenario.

1.2 Integration with Query Processor

The performance study in [4] was limited to investigating the relative performance of the CIQE suite of algorithms for various alternative settings of the design parameters. This information does not suffice for incorporation of iceberg queries in a *query optimizer* since it is not clear under what circumstances CIQE should be chosen as opposed to other alternatives. For example, questions like: At what estimated sizes of the "tip" should a query optimizer utilize CIQE? Or, what is the minimum data skew factor for CIQE to be effective for a wide range of query selectivities?, and so on, need to be answered. A related issue is the following question: Even for those environments where CIQE is applicable and does well, is there a significant difference between its performance and that of an offline optimal? That is, how *efficient* is CIQE?

1.3 Our Work

We attempt to address the above-mentioned questions in this paper. First, we place CIQE's performance for iceberg queries in perspective by (empirically) comparing it against three benchmark algorithms: **SMA**, **HHA**, and **ORACLE** over a variety of datasets and queries. In these experiments, we stop at 10% query selectivity (in terms of the number of distinct targets in the result set) since it seems reasonable to expect that this would be the limit of what could truly be called an "iceberg query" (this was also the terminating value used in [4]). SMA and HHA represent the classical approaches described above, and provide a viability bound with regard to the minimal performance expected from CIQE. ORACLE, on the other hand, represents an optimal, albeit practically infeasible, algorithm that is *apriori* "magically" aware of the identities of the result groups and only needs to make one scan of the database in order to compute the explicit counts of these qualifying groups[2]. Note that this aggregation is the *minimum work* that needs to be done by *any* practical iceberg query algorithm, and therefore the performance of ORACLE represents a lower bound.

Second, we provide a simple "recipe" algorithm that can be implemented in a query optimizer to enable it to make a decision about the appropriate algorithmic choice to be made for an iceberg query, that is, when to prefer CIQE over the classical approaches implemented in database systems. The recipe algorithm takes into account both the query characteristics and the underlying database characteristics.

[2] Since the result set is small by definition, it is assumed that counters for the entire result set can be maintained in memory.

Due to space limitations, we only focus on High-Iceberg queries in this paper – for the extensions to Low-Iceberg queries, we refer the reader to [10].

2 Algorithms for High-Iceberg Queries

As formulated in [4], a prototypical High-Iceberg query on a relation $I(target_1, ..., target_k, rest)$ and a threshold T can be written as:

```
SELECT target_1, ..., target_k, agg_function(measure)
FROM I
GROUP BY target_1, ..., target_k
HAVING agg_function(measure) > T
```

where the values of $target_1, ..., target_k$ identify each group or *target*, while *measure* ($\subseteq rest$) refers to the fields on which the aggregate function is being computed, and the relation I may either be a single materialized relation or generated by computing a join of the base relations.

We describe, in the remainder of this section, the suite of algorithms – `SMA`, `HHA`, `CIQE` – that can be used for computing High-Iceberg queries, as also the optimal `ORACLE`. For ease of exposition, we will assume in the following discussion that the aggregate function is COUNT and that the grouping is on a single attribute. Further, we will use, following [4], the term "heavy" to refer to targets that satisfy the threshold criteria, while the remaining are called "light" targets.

2.1 The SMA Algorithm

In the SMA algorithm, relation I is sorted on the target attribute using the optimized `Two-Phase Multi-way Merge-Sort` [5]. The two important optimizations used are: (a) *Early Projection* – the result attributes are projected *before* executing the `sort` in order to reduce the size of the database that has to be sorted, and (b) *Early Aggregation* – the aggregate evaluation is pushed into the `merge` phases, thereby reducing the size of data that has to be merged in each successive `merge` iteration of `external merge-sort`.

2.2 The HHA Algorithm

In the HHA algorithm, aggregation is achieved through hybrid hashing on the grouping attributes. Hybrid hashing combines in-memory hashing and overflow resolution. Items of the same group are found and aggregated when inserting them into the hash table. Since only output items are kept in memory, a hash table overflow occurs only if the output does not fit into memory. However, if an overflow does occur, partition files are created. The complete details of the algorithm are available in [7].

2.3 The CIQE Algorithm

We now describe the `CIQE` algorithms. In the following discussion, we use the notation H and L to denote the set of heavy and light targets respectively. The `CIQE` algorithms

first compute a set F of potentially heavy targets or "candidate set", that contains as many members of H as possible. When $F - H$ is non-empty, it means that there are *false positives* (light values are reported as heavy), whereas when $H - F$ is non-empty it means that there are *false negatives* (heavy targets are missed). The algorithms suggested in [4] use combinations of the following sequence of building blocks in a manner such that all false positives and false negatives are eventually removed.

Scaled-Sampling: A random sample of size s tuples is taken from I. If the count of each target, scaled by N/s, where N is the number of tuples in I, exceeds the specified threshold, the target is part of the candidate set F. This step can result in both false positives and false negatives.

Coarse-Count: An array $A[1..m]$ of m counters and a hash function h, which maps the target values from $log_2 t$ to $log_2 m$ bits, $m \ll t$, is used here. Initially all the entries of the array are set to zero. Then a linear scan of I is performed. For each tuple in I with target v not in F, the counter at $A[h(v)]$ is incremented. After completing this hashing scan of I, a bitmap array $B[1..m]$ is computed by scanning through the array A and setting $B[k]$ to one if $A[k] > T$. This step removes all false negatives, but might introduce some more false positives.

Candidate-Selection: Here the relation I is scanned, and for each target v whose $B[h(v)]$ entry is one, v is added to F.

Count: After the final F has been computed, the relation I is scanned to explicitly count the frequency of the targets in F. Only targets that have a count of more than T are output as part of the query result. This step removes all false positives.

Among the `CIQE` algorithms, we have implemented `Defer-Count` and `Multi-Stage`, which were recommended in [4] based on their performance evaluation. A brief-description of these algorithms is provided next.

Defer-Count The `Defer-Count` algorithm operates as follows: First, compute a small sample of the data. Then select the f most frequent targets in the sample and add them to F, as these targets are likely to be heavy. Now execute the hashing scan of `Coarse-Count`, but do not increment the counters in A for targets already in F. Next perform `Candidate-Selection`, adding targets to F. Finally remove false positives from F by executing `Count`.

Multi-Stage The `Multi-Stage` algorithm operates as follows: First, perform a sampling scan of I and for each target v chosen during the sampling scan, increment $A[h(v)]$. After sampling s tuples, consider each of the A buckets. If $A[i] > T * s/N$, mark the i^{th} bucket to be potentially heavy. Now allocate a common pool of auxiliary buckets $B[1..m']$ of m' ($< m$) counters and reset all the counters in A to zero. Then perform a hashing scan of I as follows: For each target v in the data, increment $A[h(v)]$ if the bucket corresponding to $h(v)$ is not marked as potentially heavy. If the bucket is so marked, apply a second hash function h' and increment $B[h'(v)]$. Next perform `Candidate-Selection`, adding targets to F. Finally remove false positives from F by executing `Count`.

2.4 The ORACLE Lower Bound Algorithm

We compare the performance of the above mentioned practical algorithms against ORACLE which "magically" knows in advance the identities of the targets that qualify for the result of the iceberg query, and only needs to gather the counts of these targets from the database. Clearly, any practical algorithm will have to do at least this much work in order to answer the query. Thus, this optimal algorithm serves as a lower bound on the performance of feasible algorithms and permits us to clearly demarcate the space available for performance improvement over the currently available algorithms.

Since, by definition, iceberg queries result in a small set of results, it appears reasonable to assume that the result targets and their counters will all fit in memory. Therefore, all that `ORACLE` needs to do is to scan the database once and for each tuple that corresponds to a result target, increment the associated counter. At the end of the scan, it outputs the targets and the associated counts.

3 Performance Evaluation for High-Iceberg Queries

In this section, we place `CIQE`'s performance in *quantitative* perspective by comparing it against the three benchmark algorithms: `SMA`, `HHA` and `ORACLE`, over a variety of datasets. We implemented all the algorithms in C and they were programmed to run in a restricted amount of main-memory, fixed to 16 MB for our experiments. The experiments were conducted on a PIII, 800 MHz machine, running Linux, with 512 MB main-memory and 36 GB local SCSI HDD. The OS buffer-cache was flushed after every experiment to ensure that caching effects did not influence the relative performance numbers.

The details of the datasets considered in our study are described in Table 1. `Dataset` refers to the name of the dataset, `Cardinality` indicates the number of attributes in the `GROUP BY` clause, `NumTargets` indicates the total number of targets in the data, `Size of DB` indicates the size of the dataset, `Record Size` indicates the size of a tuple (in bytes), `Target Size` indicates the size of the target fields (in bytes), `Measure Size` indicates the size of the measure fields (in bytes), `Skew` (measured using $LexisRatio = Var/Mean$) is a measure of the skew in the count distribution, and `Peak Count` represents the peak target count.

Dataset	Cardinality	NumTargets	Size of DB	Record Size	Target Size	Measure Size	Skew	Peak Count
D_1	1	10M	1GB	16	4	4	1657	194780
D_2	2	62M	1GB	16	8	4	1541	194765
D_3	1	8.38M	1GB	16	4	4	1.27	24
D_4	2	16.4M	1GB	16	8	4	0.89	18

Table 1. Statistics of the datasets

We now move on to the performance graphs for these datasets, which are shown in Figures 1(a)– 1(d). In these graphs, the query response times of the different algo-

rithms are plotted on the Y axis for different values of result selectivity ranging from 0.001% to 10% on the X axis (note that the X axis is on a *log scale*.) We stopped at the 10% selectivity value since it seemed reasonable to expect that this would be the limit of what could truly be called an "iceberg query" (this was also the terminating value used in [4]). Since we found little difference in the relative performance of `Defer-Count` and `Multi-Stage` for all our datasets, we have given the performance of the `Defer-Count` algorithm under the generic name `CIQE` in the graphs. In the following discussion, *low* number of targets means that for the amount of main-memory available, the average occupancy per bucket in `CIQE` algorithms is less than 5. Else we say the number of targets is *high*.

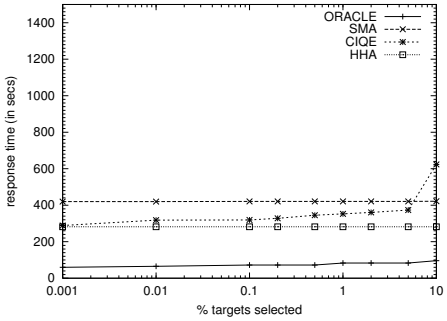

Fig. 1(a). High skew/low number of targets

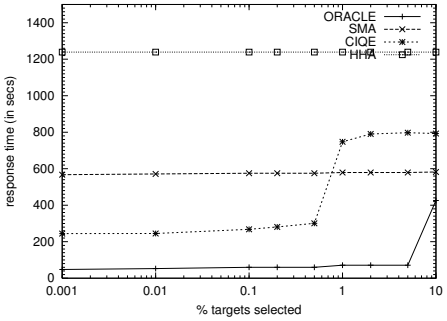

Fig. 1(b). High skew/high number of targets

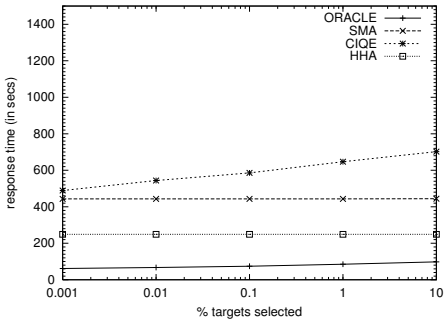

Fig. 1(c). Low skew/low number of targets

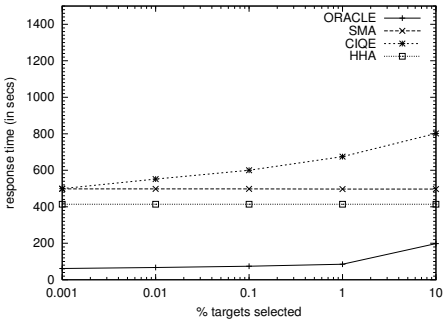

Fig. 1(d). Low skew/high number of targets

3.1 High skew, Low number of targets

Figure 1(a) corresponds to Dataset D_1 wherein the data has high skew and low number of targets, corresponding to the "favorite" scenario for `CIQE`. Therefore, as expected,

CIQE performs better than SMA for a substantial range of selectivity values (upto 7.0%). This is essentially because the average bucket occupancy (t/b) is low (= 2.75) and the peak target counts are much higher than the mean target count. However, the best overall performer is HHA, as the total number of targets are not huge compared to the number of targets that can fit in the constrained memory. Note that both SMA and HHA are unaffected by the query selectivity, unlike CIQE. Finally, we see that there is a significant gap (order of magnitude) between the performance of ORACLE and the online algorithms indicating that there appears to be some scope for designing better iceberg query processing algorithms.

3.2 High skew, High number of targets

Figure 1(b) corresponds to Dataset D_2 wherein the data has high skew with high number of targets. For this dataset, CIQE performs better than SMA for a much smaller spread of selectivity values (only upto 0.7%). This is because the average bucket occupancy in this case is almost 17, which is rather high. HHA performs worse compared to other algorithms as the number of targets are far greater than the number of targets that can fit in memory. The reason that ORACLE shows a steep increase at 10% selectivity is that the result targets exceed the available main memory.

3.3 Low skew, Low number of targets

Figure 1(c) corresponds to Dataset D_3 wherein the data has low skew with low number of targets (similar to the Dean's Query in the Introduction). Note the dramatic change in performance from Figure 1 – we now have CIQE *always* performing worse than SMA. This is entirely due to the fact that the low skew means that a significant fraction of the bits in the bit-vector turn out to be 1, effectively nullifying the desired filtering effect of the Coarse-Count step. In fact, the bit-vector had over 25% of 1's even at the highest selectivity (0.0001%). The best overall performer is HHA, due to the low number of targets.

3.4 Low skew, High number of targets

Finally, Figure 1(d) corresponds to Dataset D_4 wherein the data has low skew with high number of targets, corresponding to the "nightmare" scenario for CIQE. Therefore, not surprisingly, we see here that CIQE *always* performs much worse than SMA because the combination of the low skew and the high bucket occupancy results in completely nullifying the pruning from the Coarse-Count step. The best overall performer is again HHA.

An important point to note from the above experiments is that, apart from being stable across all selectivities, the performance of SMA is *always* within a factor of two of CIQE's performance. This means that SMA is quite competitive with CIQE. On the other hand, the performance of HHA degrades considerably as the number of targets increase. Other issues with HHA are:

– HHA opens multiple files for storing the overflow buckets on disk. This creates a problem with respect to system configuration, as there is a limitation on the number of files that can be opened by a single process. A related problem is the memory space consumed by open file descriptors [11].
– As the number of attributes in the GROUP-BY increase, it is difficult to estimate the number of targets, which is critical for choosing HHA for iceberg query evaluation.

4 Recipe Algorithm

In this section, we describe a simple "recipe" algorithm (Figure 2) that can be implemented in the query optimizer to enable it to make a decision about the appropriate algorithmic choice to be made for a High Iceberg query, that is, whether to choose CIQE or SMA. We do not consider HHA here, because as discussed at the end of Section 3, HHA is not suitable for the kind of datasets (within the given memory constraints) we consider here.

For Iceberg queries involving the AVERAGE, MIN or MAX aggregate functions, SMA is the only choice among the suite of algorithms we consider here since CIQE/MINI pruning techniques do not work for these functions.

For High-Iceberg queries involving COUNT or SUM on a single relation, we make a binary decision between SMA and CIQE based on the conditional in the formula on line 9. Estimating the $total$ mentioned in this formula is simple and is done the same way as in Scaled-Sampling. i.e. compute the total for the sample size s ($total_{sample}$) and then scale it to the dataset size by multiplying by N/s.

DS	T_{act}	S_{act}	T_{est}	S_{est}
D_1	12	7.00	18	4.40
D_2	19	0.0002	18	0.0004
D_3	72	0.70	18	1.02
D_4	18	0.0001	18	0.0001

Table 2. Crossover point : actual vs estimated

We verified the accuracy of this binary decision for the datasets involved in our study. Table 2 presents a summary of these results. In this table, T_{act} refers to the actual threshold (based on the experiments) below which SMA starts performing better than CIQE, S_{act} refers to the corresponding percentage target selectivity, T_{est} refers to the estimated threshold (based on the formula) below which SMA should start performing better than CIQE, and S_{est} refers to the corresponding percentage target selectivity. As shown in the table, the selectivity estimates where SMA will start performing better than CIQE are very close to the numbers from the experimental study.

So far, we had implicitly assumed that the iceberg query was being evaluated over a single base relation. But in case of an iceberg query involving a join of multiple base relations, the iceberg relation I is derived from the base relations B using one of the efficient join algorithms such as, for example, sort-merge join or hybrid-hash

```
Iceberg_Query_Optimizer_Module (B, G, J, A, T, M)
Input:
    B - set of relations in the query i.e. FROM clause,
    G - set of attributes in the group-by i.e GROUP BY clause,
    J - set of attributes in the equi-join i.e. WHERE clause,
    A - aggregate function on the targets,
    T - threshold on the aggregate function i.e. HAVING clause,
    M - memory for computing the query
Output:
    C - choice of algorithm to use for computing the Iceberg Query
        CIQE, SMA, SA.
1. if (A = AVERAGE or MIN or MAX) // irrespective of whether O is '<' or '>'
2.     return SMA
3. if (A = COUNT or SUM)
4.     if (|B| = 1) // single relation
5.         b = number of hash buckets for CIQE in the available memory M
6.         if (A = COUNT)
7.             total = N
8.         else
9.             Sample B
10.                Estimate total = aggregate value treating the whole database
                       as a single target = N/s * total_sample
11.                if (total/b < T) // takes care of average per bucket occupancy,
                       skew and selectivity
12.                    return CIQE
13.                else
14.                    return SMA
15.    else if (|B| > 1) // join of multiple relations
16.        if (J ∩ G = φ) // no interesting join order possible
17.            D = the amount of free disk space
18.            Estimate S = the size of the join
19.            if (S < 2 * D)
20.                // same as |B| = 1 above
21.            else
22.                return CIQE
24.        else if (J ∩ G ⊂ G and join output sorted on J)
25.            if (grouping on attributes in G − J for the individual targets
                   based on attributes in J can be done in memory)
26.                return SA
27.            else
28.                return CIQE
29.        else if (J ∩ G = G and join output sorted on J)
30.            return SA
```

Fig. 2. Recipe Algorithm

join [7]. For the case where the group-by clause shares some attributes with the join attributes, the query optimizer may opt for join algorithms that produce "interesting orders" ([7],[14]) – that is, where the output is sorted on one or more attributes. As a result of this, the sorted tuples from the result of the join can be piped (using the iterator model discussed in [7]) to the following aggregate operation, which can then aggregate the tuples in memory to produce the final query result. We use Simple Aggregation (SA) to refer to such a situation where the aggregation is computed on pre-sorted output, and this SA technique is also incorporated in the concluding part of the recipe algorithm (Figure 2).

5 Related Work

Apart from the CIQE set of algorithms [4] previously discussed in this paper, there is comparatively little work that we are aware of that deals directly with the original problem formulation. Instead, there have been quite some efforts on developing *approximate* solutions (e.g. [1, 6, 9, 12]). In [12], a scheme for providing quick approximate answers to the iceberg query is devised with the intention of helping the user refine the threshold before issuing the "final" iceberg query with the appropriate threshold. That is, it tries to eliminate the need of a domain expert or histogram statistics to decide whether the query will actually return the desired "tip" of the iceberg. This strategy for coming up with the right threshold is complementary to the efficient processing of iceberg queries that we consider in this paper.

As mentioned before, the CIQE algorithm works only for simple COUNT and SUM aggregate functions. Partitioning algorithms to handle iceberg queries with AVERAGE aggregate function have been proposed in [2]. They propose two algorithms, BAP (Basic Partitioning) and POP (POstponed Partitioning) which partition the relation logically to find candidates based on the observation that for a target to satisfy the (average) threshold, it must be above the threshold in at least one partition. The study has two drawbacks: First, their schemes require writing and reading of candidates to and from disk, which could potentially be expensive, especially for low skew data. Second, their performance study does not compare BAP/POP with respect to SMA, making it unclear as to whether they are an improvement over the current technology. In our future work, we plan to implement and evaluate these algorithms.

All the above work has been done in the context of High-Iceberg queries. To the best of our knowledge, there has been no prior investigation of Low-Iceberg queries which appears in the technical report version [10] of this paper.

6 Conclusions

In this paper, we have attempted to place in perspective the performance of High-Iceberg query algorithms. In particular, we compared the performance of CIQE with regard to three benchmark algorithms – SMA, HHA and ORACLE – and found the following:

- `CIQE` performs better than `SMA` for datasets with low to moderate number of targets, and moderate to high skew. It never performs better than `SMA` for datasets with low skew and high number of targets.
- The performance of `CIQE` is never more than twice better than that of `SMA` for the cases where the base relation is materialized and there is enough disk space to sort the relation on disk.
- While `HHA` did perform well in several cases, its performance was not robust in that it could perform quite badly when the number of targets was high, and in addition, it has implementation difficulties.
- There was a considerable performance gap between the online algorithms and ORACLE, indicating a scope for designing better iceberg query processing algorithms.

We also described a simple recipe algorithm for the incorporation of Iceberg queries in the Query Optimizer. This recipe takes into account the various data and query parameters for choosing between classical and specialized techniques.

Acknowledgements This work was supported in part by a Swarnajayanti Fellowship from the Dept. of Science & Technology, Govt. of India.

References

1. "AQUA Project", *http://www.bell-labs.com/project/aqua/papers.html*.
2. J. Bae and S.Lee, "Partitioning Algorithms for the Computation of Average Iceberg Queries", *Proc. of DAWAK Conf.*, 2000.
3. D. Bitton and D. Dewitt, "Duplicate Record Elimination in Large Data Files", *ACM Trans. on Database Systems*, 8(2):255–265, 1983.
4. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani and J. Ullman, "Computing Iceberg Queries Efficiently", *Proc. of 24th Intl. Conf. on Very Large Data Bases*, 1998.
5. H. Garcia-Molina, J. Ullman, and J. Widom, "Database System Implementation", *Prentice Hall*, 2000.
6. A. Gilbert, Y. Kotidis, S. Muthukrishnan and M. Strauss, "Surfing wavelets on streams: one-pass summaries for approximate aggregate queries,", *Proc. of 27th Intl. Conf. on Very Large Data Bases*, 2001.
7. G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Comput. Surv.*, 25, 2, 73–170, June 1993.
8. Y. Ioannidis and V. Poosala, "Histogram-Based Solutions to Diverse Database Estimation Problems", *IEEE Data Engineering*, Vol. 18, No. 3, pp. 10-18, September 1995.
9. I. Lazaridis and S. Mehrotra, "Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure", *Proc. of ACM SIGMOD Conf.*, 2001.
10. K. Leela, P. Tolani and J. Haritsa, "On Incorporating Iceberg Queries in Query Processors", *Tech. Rep. TR-2002-01*, DSL/SERC, Indian Institute of Science, February 2002.
11. http://linuxperf.nl.linux.org/general/kerneltuning.html
12. Y. Matias and E. Segal, "Approximate iceberg queries", *Tech. Rep.*, Dept. of Computer Science, Tel Aviv University, Tel Aviv, Israel, 1999.
13. R. Ramakrishnan and J. Gehrke, "Database Management Systems", *McGraw-Hill*, 2000.
14. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price, "Access Path Selection in a Relational Database Management System", *Proc. of ACM SIGMOD Conf.*, 1979.