ON INCORPORATING ICEBERG QUERIES IN QUERY PROCESSORS

Krishna P. Leela Pankaj M. Tolani Jayant R. Haritsa

Technical Report
TR-2002-01

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

On Incorporating Iceberg Queries in Query Processors

Krishna P. Leela Pankaj M. Tolani Jayant R. Haritsa*

Database Systems Lab, SERC

Indian Institute of Science

Bangalore 560012, India

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.

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 in Example Query 1 for the Relation REGISTER(RollNo, CourseID, Credits, Grade):

Example Query 1:

```
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

*Contact Author: haritsa@dsl.serc.iisc.ernet.in

query is called an **iceberg query** [1] – 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:

1. 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
2. the relation is recursively *partitioned* using hash functions resulting in partitions in which the distinct groups fit in the available main memory and are hence grouped in main memory

In general, these strategies appear wasteful since they do not take the threshold predicate into account, that is, they are not output sensitive. 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: `sort-merge join`, `hybrid-hash join`, and others mentioned in [2]. For the case where the group-by clause shares some attributes with the join attributes, the query optimizer opts for algorithms that produce "interesting" orders ([2],[3]). As a result of this, the tuples from the result of the join can be piped (using the iterator model discussed in [2]) to the following aggregate operation, which can then aggregate the tuples in memory to produce the final query result. If such an "interesting" order is not possible for the result of join, `I` needs to be computed followed by one of the two approaches mentioned for a single relation above. We consider only the latter case in the rest of the paper for reasons mentioned in Section 2.

Motivated by the above observation, a variety of customized algorithms for efficiently handling iceberg queries were proposed and evaluated in [1] 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.

`CIQE` represents the pioneering work in defining and tackling iceberg queries. A particularly attractive feature of `CIQE` is that it does not entail *materializing* the join of the relations on which the GROUP-BY is being applied, and is therefore not affected by the size of the join of the relations. However, it still has significant lacunae, described in more detail below, with regard to its scope of applicability and its integration with query processors. These issues need to be addressed before specialized iceberg query processing can become a viable proposition in real systems – we take the first step in doing so in this paper.

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

1.1 Scope of Applicability

CIQE can be utilized only in a restricted set of iceberg query environments – specifically environments in which

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

An implication of the last constraint (high skew) is that CIQE would *not work* for Example Query 1 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, as shown in Example Query 2. 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.

Example Query 2:

```
SELECT RollNo
FROM REGISTER
GROUP BY RollNo
HAVING MIN(Grade) > 'B'
```

Finally, the impact of the first 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 alternative version of the "Dean's Query" could be to find the part-time students who are taking *less than* 6 credits, as shown in Example Query 3:

Example Query 3:

```
SELECT RollNo, SUM(Credits)
FROM REGISTER
GROUP BY RollNo
HAVING SUM(Credits) < 6
```

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

[2] For ease of exposition, we use $>$ and $<$ to also imply $\geq$ and $\leq$, respectively.

a *much harder* problem since there are no known efficient techniques to identify the lowest frequencies in a distribution [7]. 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 [1] 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, at 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 the best possible (if quantifiable), encouraging researchers to try and devise even better algorithms? That is, how *efficient* is CIQE?

1.3 Our Work

We attempt to address the above-mentioned limitations and 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 [1]). SMA and HHA represent the classical approaches described above, and provide a viability bound with regard to the minimal performance expected from CIQE. Note that even though the list of classical algorithms we consider here for computing the iceberg queries is not exhaustive, existence of better algorithms in specific scenarios only means that the performance gap between classical algorithms and CIQE can only *reduce* in comparison to the results shown in the experimental section. Finally, ORACLE 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[3]. 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 (for the first time) a customized algorithm, called MINI, to handle Low-Iceberg queries. MINI is a multi-pass algorithm that in each pass over the database, partitions it into memory-sized partitions and generates a new mini-database that is compressed both vertically and horizontally. The mini-database forms the input to the following pass, and the iteration ends when the mini-database becomes small enough to

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

completely fit in memory. We compare MINI's performance against the same suite of benchmark algorithms described above, i.e., SMA, HHA and ORACLE.

Finally, 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 or MINI over the classical approaches implemented in database systems. The recipe algorithm takes into account both the query characteristics and the underlying database characteristics.

1.4 Contributions

To summarize the contributions of this paper:

- We provide a complete performance framework for the CIQE algorithm.
- We define Low-Iceberg queries and present a new algorithm for efficiently processing such queries.
- We provide a recipe algorithm, suitable for inclusion in a query optimizer, to make the appropriate algorithmic choice for High and Low iceberg query execution.

1.5 Organization

The remainder of this paper is organized as follows: In Section 2, we focus on High-Iceberg queries and describe the suite of algorithms (SMA, HHA, CIQE and ORACLE). The performance of these algorithms is presented in Section 3. We then move on to defining Low-Iceberg queries and presenting our new MINI algorithm in Section 4. The performance of MINI with respect to SMA, HHA and ORACLE is evaluated in Section 5. Next, in Section 6, we provide the recipe algorithm intended for use in the query optimizer. In Section 7, we overview the related work, and finally, in Section 8, we summarize the conclusions of our study and outline future avenues to explore.

2 High-Iceberg Queries

Our focus in this section is on High-Iceberg queries, that is, where the threshold represents a lower bound on the aggregate value of result groups. As formulated in [1], 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. We also discuss the impact of various data and query parameters on the performance of this suite of algorithms. 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. Also as mentioned in the Introduction, the iceberg query is assumed to be executed either on a single relation or on a join of relations where "no" interesting join order is possible. For the cases where the join of relations can be produced in an interesting order, a simple sequential in memory aggregate after the join can be used to compute the iceberg query. Further, we will use, following [1], the term "heavy" to refer to targets that satisfy the threshold criteria, while the remaining are called "light" targets.

2.1 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: the result attributes are projected *before* executing the `sort` in order to reduce the size of the database that has to be sorted, and 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`. From the analysis shown in [4], performance of SMA is not linear in the size of data, and requires number of merge passes proportional to the logarithm of the data size. But for sufficient main-memory, SMA finishes in 3-5 passes of the database for most real-world dataset sizes.

2.2 The HHA Algorithm

Algorithms based on hybrid hashing can also be used for aggregation by 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, not input items, are kept in memory, hash table overflow occurs only if the output does not fit into memory. However, if overflow does occur, partition files are created. Thus, hybrid hashing determines dynamically how much input data truly must be written to temporary disk files. All partition files in any one recursion level are as large as the entire input because once a partition is written to disk, no further aggregation can occur until the partition files are read back into memory. Details of the algorithm and its complexity can be found in [2].

2.3 The CIQE Algorithm

We now describe the CIQE algorithms and then discuss their performance at an intuitive level. 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 [1] 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.

The analysis of the above steps suggests that `CIQE` performance is linear in the size of the data as long as the final F fits in memory. For the case where the final F does not fit in memory, `Count` resorts to `SMA` on a filtered database containing only tuples corresponding to the targets in F, also generated during `Candidate-Selection`.

Among the `CIQE` algorithms, we have implemented `Defer-Count` and `Multi-Stage`, which were recommended in [1] based on their performance evaluation. A brief-description of these algorithms is provided next.

2.3.1 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`.

2.3.2 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.

2.5 CIQE versus SMA/HHA

We conclude this section with an informal discussion of the *qualitative* impact of the different data and query parameters on the relative performance of `SMA`, `HHA` and `CIQE`. In particular, we consider the following three parameters: the number of targets (t), the mean (m) of the aggregate values (i.e. counts) of the targets, and the result selectivity (s) as reflected by the threshold T.

2.5.1 Number of Targets

For a fixed amount of memory, this parameter really does not affect `SMA`, since the performance of both `sort` and `merge` are only mildly dependent on the number of targets. On the other hand, as the number of targets play a critical role in deciding the number of partitions in HHA, the increase results in a considerable degradation in the performance of `HHA`. Also the performance of `CIQE` degrades considerably with an increase in the number of targets. This is because more and more targets collide to the same hash bucket, and as a result, `Coarse-`

Count results in more and more false positives. Eventually, after some point, not all the targets in F can fit in memory, and Count resorts to SMA for computing the iceberg query.

2.5.2 Mean Target Count

This parameter does not affect SMA since sort and merge perform the same number of comparisons and hence the same amount of work, independent of the distribution of the target counts. But skew in the distribution of the target counts contributes to the reduced performance of HHA. This is because the hash function fails to produce nearly equi-sized partitions in that case. Also it affects the performance of CIQE considerably since Coarse-Count's pruning ability is dependent on this parameter, as explained below.

For target count distributions with high skew, that is, where $m \ll T$, Coarse-Count works fine since the combination of many light targets colliding to the same hash bucket still do not make these buckets to appear as heavy. As a result, this step results in eliminating many light targets, and hence a small $|F|$ for Count. On the other hand, for target count distribution with low skew ($m \sim T$), even a few light targets colliding to the same bucket make its occupants appear as heavy. As a result, $|F|$ becomes large, comparable to the total number of targets in the relation, forcing Count to resort to SMA as its exit policy.

2.5.3 Result Selectivity

This parameter mildly affects the aggregate phase of SMA, which decides the number of targets that qualify and hence the amount of work to be done in outputting the query result. Same if true for HHA. In contrast, this parameter affects the performance of CIQE considerably. This is because a decrease in selectivity implies a lower threshold value, which means Coarse-Count will become less effective in eliminating light targets for the final F, and as a result the final F may not fit in memory.

2.5.4 Summary

To summarize the above discussion, we present here a simple metric to guide us about when CIQE might be expected do well. Denoting the average number of targets hashing to each bucket as (t/b) and the mean target count of m, the average "weight" of each bucket is $(t/b) * m$. When this is normalized to the threshold T, we get the "normalized average bucket weight" ($NABW$) equal to $(t/b) * (m/T)$. Now, if $NABW < 1$, we expect that CIQE will do well since F will be small.

3 Performance Evaluation for High-Iceberg Queries

Moving on from the qualitative comparison of the previous section, we now 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/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. Again, as mentioned in the Introduction, we do not consider here the cases where interesting join orders are possible.

Data-set	Cardi-nality	Num-Targets	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 algorithms are plotted on the Y axis for different values of result selectivity ranging from 0.001% to 10% on the X axis. (Note that both the X axis uses *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 [1]). 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*.

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. Also note that, as discussed in Section 2.5.3, both `SMA` and `HHA` are unaffected by

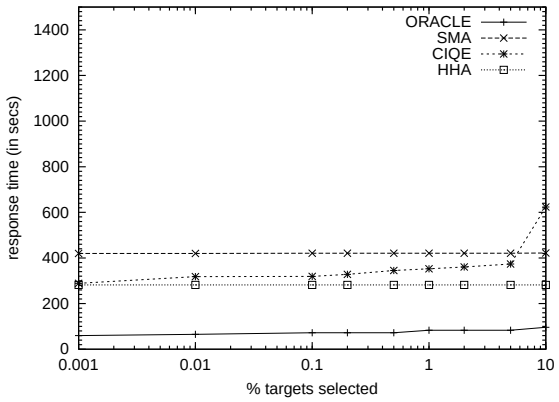

Figure 1(a): **High skew/low number of targets**

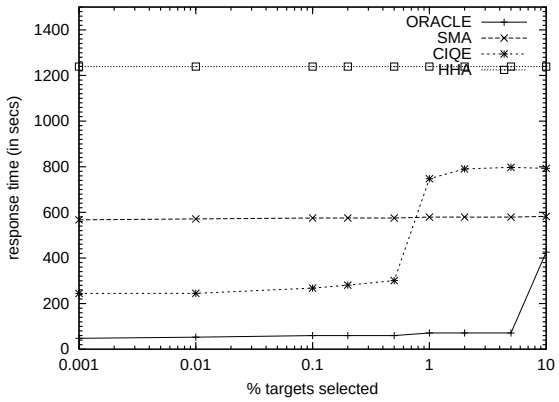

Figure 1(b): **High skew/high number of targets**

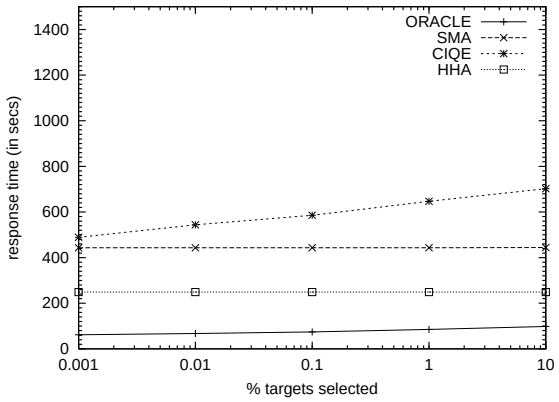

Figure 1(c): **Low skew/low number of targets**

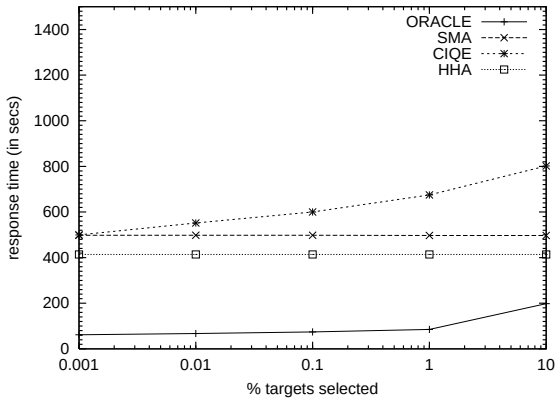

Figure 1(d): **Low skew/high number of targets**

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 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 lower spread of selectivity values (only upto 0.7%). This is because the average bucket occupancy is 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. Again note that, as discussed in Section 2.5.3, both `SMA` and `HHA` are unaffected by the query selectivity, unlike `CIQE`. 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%). Again, 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.

3.4 Low skew, High number of targets

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. Again, 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. The reason that ORACLE shows a steep increase at 10% selectivity is that the result targets exceed the available main memory at this selectivity.

Another 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. Another problem is the memory space consumed by open file descriptors [20].
- 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 Low-Iceberg Queries

The previous sections dealt exclusively with High-Iceberg queries. We now move on to considering *Low-Iceberg* queries, which compute aggregate functions *below* a user-specified threshold, and present a customized algorithm, called MINI, for efficiently evaluating such queries using compact main-memory structures. To the best of our knowledge, there has not been any prior work in this area.

At first glance, one might think that it is not necessary to formulate the Low-Iceberg query problem separately and that some minor variation of the specialized algorithms for High-Iceberg queries would easily apply to Low-Iceberg queries as well. But, as we explain below, none of the techniques developed in CIQE work for Low-Iceberg queries, since their use would lead to false negatives, processing which will be as difficult as the original problem.

- `Scaled-Sampling` in `CIQE` helps in a quick probabilistic search of the high-frequency targets that are likely to belong to the final answer in the High-Iceberg query case. However, in the case of Low-Iceberg queries, the low-frequency targets belonging to the final result will *not* show up because they occur too infrequently in the database.

- The pruning technique used in the `Coarse-Count` step of `CIQE` cannot be used for Low-Iceberg queries. It would be incorrect to prune away targets belonging to the buckets that have a count exceeding the threshold, as the individual counts may still be lower than the threshold. On the plus side, all targets hashing to a bucket that turns out to be light are *guaranteed* to be light. But, in practice, only a few buckets, if any at all, will exhibit this feature due to the large number of heavy targets present.

On the other hand, note that both `SMA` and `HHA` are capable of handling Low-Iceberg queries since they operate independent of the threshold constraint.

4.1 MINI

In this section, we propose a simple, multi-pass partition-based algorithm, `MINI`, for evaluating Low-Iceberg queries. A partition $P \subseteq$ relation R refers to any contiguous subset of tuples contained in R and union of all partitions is the relation, i.e., $P_1 \cup P_2 \cup \ldots \cup P_n = R$. The key idea underlying the `MINI` algorithm is to partition the data and prune the high ranked heavy targets (targets whose aggregated measure ranks high among all heavy targets) from the database at each pass. We now present the theorem used for pruning the candidates. In the following text, terms 'local' and 'global' count represent the count of a target within a partition and for the whole database respectively.

Theorem Given threshold T and relation R, partitioned into n partitions ($P_1, P_2, ..., P_n$),

$\forall\ targets\ t\ \in R$

$$(\exists\ i\ local\_count(t, P_i)) \geq T \Rightarrow global\_count(t, R) \geq T \qquad i = 1,2,...,n;$$

The theorem states that if a target is heavy in at least one partition P, then it must be heavy with respect to the whole relation R. Therefore, we can prune those targets that are beyond the threshold in each of the partitions.

At the end of every pass, we retain some of the high ranked heavy targets, which constitute most of the database. We also prune some low ranked heavy targets when these are replaced by some high rank heavy targets. And a new mini-database is created, which is composed of targets that proved to be locally light in some partition. This database, which we refer to as the mini-database, will be smaller in size as compared to the input database in this iteration, and is fed to the next iteration. We discuss the proof for guaranteed reduction in Section 4.1.2. The iterative process continues until we reach a stage where all the remaining candidate targets fit in memory after which we do a counting scan of the *base* data to eliminate all the false positives. Note that no false negatives are possible in our scheme because we never remove any light targets from the database.

Apart from the elimination of tuples corresponding to globally heavy targets, the following strategies help to further reduce the effective database size:

- During the first pass over the base database, only the *result attributes* from the tuples are written to the mini-database – this is similar to the *early projection* optimization of SMA/HHA.
- During all passes, *early aggregation* of the tuples corresponding to a common target results in vertical compression of the database, again similar to that of SMA/HHA.

The architecture of the algorithm is shown in Figure 2. We briefly describe the steps of the algorithm below followed by a detailed functioning in Figure 3. Again for ease of exposition, we will assume the aggregate function being computed is COUNT, but the algorithm easily extends for SUM.

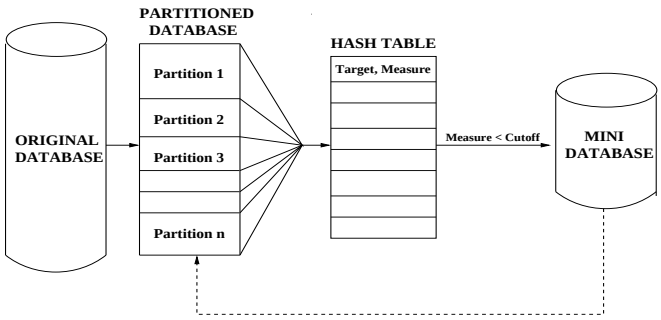

Figure 2: **Architecture of MINI**

For each tuple, hash the target to a main-memory collision-resolving hashtable (i.e. distinct targets occupy different buckets), and update the counter at the corresponding location. This continues till the hashtable becomes full which marks the end of the current virtual partition and the beginning of the next partition. Now, remove from the hashtable all the targets whose counter values are less than the `Cutoff` (shown in Figure 3) as discussed below in Section 4.1.1, and write these along with their counter values to the mini-database.

Therefore, after every pass the mini-database consists of targets that appear locally light in a partition, but are not guaranteed to be globally light (false positives). But the algorithm does not result in false negatives, as light targets are *always* written.

Note that since heavy targets are not removed from the hashtable once they are identified to be heavy (since Cutoff is always less than Threshold), after some number of partitions we might reach a stage where the entire hashtable has only heavy targets, and therefore there is no further room for removal of any targets. From this point in time, the following procedure is followed until all the tuples in the database have been processed: for each tuple, hash on the target value and if the target does not exist in the hashtable, write the tuple to the mini-database. This ensures that the tuples for the high ranked heavy targets identified during the current pass continue to be pruned, resulting in significant reductions in database size. At the end of the pass, the hashtable is flushed.

MINI ($Base\_DB, \mathcal{T}, \mathcal{M}$)
Input : Database $Base\_DB$, threshold for Low-Iceberg query $\mathcal{T}$, Memory Size $\mathcal{M}$
Output: Result Set $\mathcal{F}$ with targets less than threshold $\mathcal{T}$

```
1. DB = Base_DB
2. Mini Database Mini_DB = NULL // mini-database to store the light targets
3. while (1) // iterative calls, stop when all targets fit in memory
4.       pruned = ∞, Prior_cutoff = 0, Cutoff = 0
5.       while (pruned > 0) // till hash is full of heavy targets
           // Hashing Scan
6.          while (HT is not full)
7.                ReadNextTuple()
8.                Hash on target
9.                if (target already exists)
10.                     target.count = target.count + 1
11.               else
12.                     insert into HT
13.                     target.count = 1
14.         end while
           // Pruning Scan
15.         pruned = 0
16.         if (Prior_cutoff != T)
17.               Cutoff = Min (T, Median of all measures in the HT )
18.               Prior_cutoff = Cutoff;
19.         else
20.               Cutoff = Median of all measures less than T in HT
21.         while (1)
22.               if (Cutoff != T)
23.                     for each entry in HT
24.                           if (target.count < Cutoff)
25.                                 append to Mini_DB
26.                                 remove from HT
27.                                 pruned = pruned + 1
28.                     break while
29.               else
30.                     Cutoff = Median of all measures less than T in HT
31.         end while
32.      end while
33.      if no tuples in database // all targets fit in hash. so, stop the iterative calls.
34.         break while
           // Filtering Scan
35.      while (more tuples in database)
36.         ReadNextTuple()
37.         Hash on target
38.         if (target does not exist in HT)
39.               append to Mini_DB
40.      DB = Mini_DB;
41.      Mini_DB = NULL
42. end while
           // Counting Scan
43. for each target in HT      target.count = 0;
44. for each target in DB
45.      if target exists in HT
46.         target.count = target.count + 1
47. for each target in HT
48.      if target.count in HT < T
49.         insert target in F
```

Figure 3: **Algorithm MINI (for COUNT aggregate)**

The above iterative steps continue until all targets of mini-database fit in memory, at which point these targets are counted over the base database in order to remove all the remaining false positives.

4.1.1 Hash Table Replacement Policies

In this section, we reason out the choice of the `Cutoff` parameter shown in Figure 3, for the replacement of the hashtable entries each time the hash becomes full. This replacement criterion helps in deciding which targets should reside in the hashtable and which should be written to the mini-database at the end of the partition. It is expressed in terms of the percentage of targets that are retained to maintain their "dynamic" global count (DGC), which represents the count of a target across multiple partitions, while the target is retained in the hashtable. When a target is retained in the hashtable for the entire iteration, this equals the exact global count (EGC), which represents the count of a target for the whole database.

Zero Reserved Replacement Policy

The name 'Zero Reservation' implies that the number of targets that are retained to maintain their DGC are zero percent of the total number that can fit in memory. At the end of each partition, we prune all the entries which are locally heavy and write all those which are locally light to a mini-database (i.e. removing all the targets from the hashtable). We now evaluate the limitations of this policy.

Consider a worst case scenario of the base database, where the targets are appearing alternately in each partition. Here, even if most targets are locally light and globally heavy (considering the whole database), we will end up writing all the targets in each partition to the mini-database. So there are scenarios where this replacement policy fails to reduce the database size and `MINI` will never terminate.

Fully Reserved Replacement Policy

We now consider the other extreme, where the number of targets that are retained to maintain their DGC is equal to the total number of targets that can fit in constrained memory. At the end of each partition, as all targets in the memory are retained in the hashtable, no pruning takes place. However, at the end of an iteration over the database, we remove the targets which are heavy in the hashtable (note that DGC equals EGC for all targets in the hashtable). If the total number of targets that can fit in memory is x, then this policy is similar to removing all tuples corresponding to x number of targets, after every iteration from the database. However this policy ignores the result of theorem mentioned at the start of the section.

Now we consider a nightmare scenario for this replacement policy. Consider a case where all the low ranked heavy targets and light targets appear in initial partitions. As no pruning happens, all the low ranked heavy targets along with light targets are retained in the hash at the end of an iteration over the database. During `pruning scan`, as we are removing only the low ranked heavy targets and light targets, the size of the mini-database does not reduce much compared to the database on which we iterated, and hence `MINI` does not perform well.

Half Reserved Replacement Policy

In this policy, we retain about half of the targets to maintain their DGC. As shown in Figure 3, we use the median of the counter values to implement this policy. At the end of each partition, we write all the targets whose count is less than `Cutoff`, defined as min(median, threshold) in Figure 3, to mini-database.

We now verify whether this replacement policy provides sufficient throughput for the worst case scenarios considered for the earlier replacement policies. For database with targets appearing alternatively in partitions, we retain heavy targets which are high ranked among the partitions seen so far and this may result in hashtable having all the heavy targets after some partitions, as median will be greater than the threshold after some partitions. For the other case, where the light targets appear in initial partitions, as we use median in deciding the `Cutoff` , we may initially retain the light targets in the hashtable, but these will be pruned after the initial partitions are scanned, as we come across heavy targets which replace the light targets.

Other Replacement Policies

We also tried 25% and 75% reserved policies and inferred the following from the results: For 25% reserved policy, the performance is bad as `Cutoff` reaches threshold quickly and most of the targets in the free 75% are locally light and need to written to mini-database, resulting in smaller reductions in database size across iterations. Using 75% reserved policy does not improve the situation as hashtable fills up very quickly every time (as there is only 25% room for new targets). And as a consequence, we need to execute `pruning scan` more often and this incurs overhead. Hence, we use 'Half Reserved' replacement policy for `MINI` based on the above discussion.

4.1.2 Convergence of MINI

Convergence of `MINI` relies on the assumption that the database size reduces over iterations. Here is a proof for the convergence.

Proof: Let us assume $|R|$ = f * $|T|$ and $|S|$ = g * $|M|$, where R, T, S and M stands for result number of targets, total number of targets, result target space and memory constraint respectively, factors 'f' and 'g' represent percentages. The above formula implies that, the result number of targets will be a small percent of the total number of targets and the memory occupied by these result targets is again a small percent of total memory. For iceberg query domain, the factors 'f' and 'g' will be approximately less than 1% and 25% respectively. So, the result will actually fit in memory. We now provide two lemma's to support our proof.

Lemma 1: A target in the reserved category can only get replaced by another target whose aggregate measure is 'strictly' greater than its own.

Lemma 2: If no target in the reserved category is replaced, then some fraction of the reserved category has to be eventually heavy, since we assume that result size is approximately less than 25% of memory size.

Lemmas 1 and 2 consider the cases where a target in reserved category can be replaced and cannot be replaced respectively. According to Lemma 1, a target in reserved category can only be replaced by a target whose aggregate measure is 'strictly' greater than its own, which implies that we will retain highly ranked heavy targets, whose pruning will reduce the mini-database to quite an extent. Now, consider a worse case

situation, where no target in the reserved category can be replaced, i.e., all the targets are light. According to Lemma 2, when no target in the reserved category can be replaced, as result size is approximately less than 25% of the constrained memory size, the other fraction of reserved partition will eventually have heavy targets, which provides proof for guaranteed reduction.

Experimental results supporting this proof are shown in Figure 4 which shows convergence of MINI for both varieties of data: high-skew and low-skew.

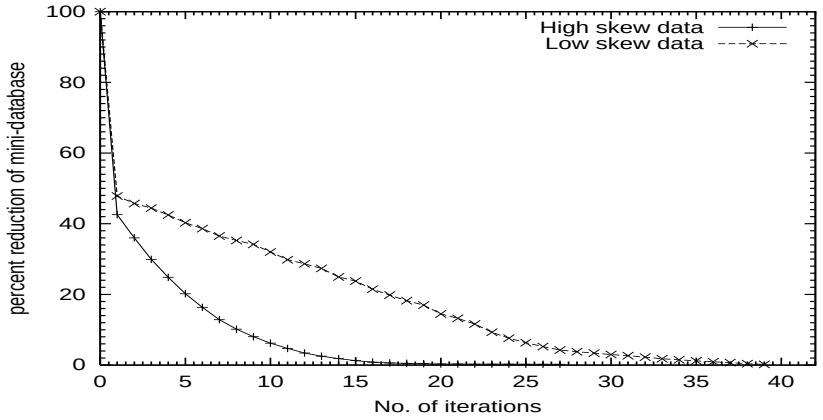

Figure 4: **Convergence of MINI for High and Low data skew**

4.2 MINI versus SMA/HHA

We now consider the same parameters discussed in Section 2.5 to study the impact of these on the performance of MINI.

4.2.1 Number of targets

As the number of targets increase, the number of false positives increase, and this results in smaller reductions in mini-database for MINI. But for Low-Iceberg queries, a small increase in heavy targets means a considerable increase in the database size. But this will not affect MINI as much as SMA, as MINI's performance scales linearly with the size of the database.

4.2.2 Mean Target Count

For target count distribution with high skew ($mean \gg T$), MINI performs better, because most of the database is occupied by high ranked heavy targets, which are efficiently pruned by the pruning scan in each pass, resulting in good reduction across iterations. For target count distribution with low skew ($mean \sim T$) , even though the number of heavy targets in each partition are small, due to the replacement policy there will be a good reduction in mini-database size.

4.2.3 Result Selectivity

This parameter decides the number of groups in the result set. As the result selectivity decreases, the number of false positives increase, thus affecting MINI's performance. But, in general, the selectivity will be high for true "iceberg" queries. As the result selectivity decreases, the result targets will not fit into the memory, resulting in MINI performing multiple scans on the base database for counting scan.

5 Performance Evaluation of MINI

In this section, we present the performance evaluation model and experimental results of MINI's performance for various datasets. We compared MINI's performance with SMA, HHA, and ORACLE. The details of the datasets considered are described in Table 2. The column names of this table are the same as those explained in Section 3. The datasets are characterized by different amount of skew in the target count distribution and the number of targets. We considered both high skew and low skew varieties in our datasets. Datasets D_5 and D_6 represent high skew data, while D_7 and D_8 represent low skew data.

When we considered the same memory constraint of 16MB as used in evaluating High-Iceberg algorithms, MINI and HHA took only one pass as the total number of targets in these datasets fit in memory. This is shown in Figure 5, which corresponds to Dataset D_1. To make the experiments tractable, we could not increase the number of targets, as this results in enormous increase in the size of the database, exceeding our disk capacities of 36GB. Hence, to make MINI take multiple passes, we lowered the memory constraint for MINI to 8KB, while using the same 16MB for others.

In the following experiments moderate number of targets means that the number of targets that can fit in the constrained memory of 8KB is only 10% of the targets in the dataset. On the other hand, high number of targets means that the number of targets that can fit in the 8KB memory are much less, in our case only 5% of the targets in the dataset. We now explain the performance graphs for our datasets shown in Figures 6(a)-6(d). Again, the Y axis represents the query response time in seconds for various result selectivities, ranging from 0.001% to 10% on the X axis.

Data-set	Cardi-nality	Num-Targets	Size of DB	Record Size	Target Size	Measure Size	Skew	Least Count
D_5	1	5000	2GB	16	4	4	40159	2
D_6	2	10000	2GB	16	8	4	16618	1
D_7	1	5000	2GB	16	4	4	83	1
D_8	1	10000	2GB	16	4	4	66	1

Table 2: **Statistics of the datasets**

5.1 High skew, Moderate number of targets

Figure 6(a) corresponds to Dataset D_5, wherein the data has high skew and moderate number of targets. For this dataset, MINI works better by about a factor of two over SMA and a factor of three over HHA. MINI performs

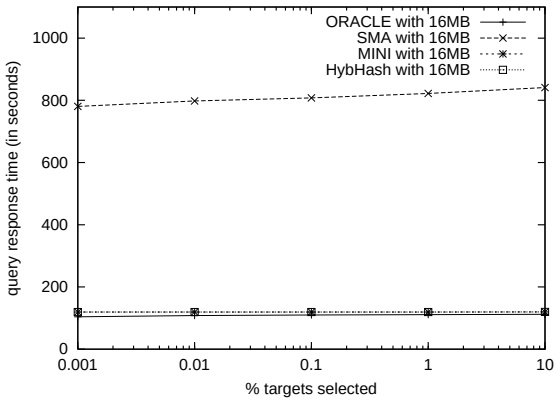

Figure 5: **High skew/moderate number of targets**

better as most of the database is occupied by the high ranked heavy targets, which are efficiently pruned by MINI's replacement policy. Note that, both SMA and HHA are unaffected by query selectivity, as it is only the aggregate phase which is affected.

5.2 High skew, High number of targets

Figure 6(b) corresponds to Dataset D_6, wherein the data has high skew and high number of targets. For this dataset, MINI performs better, even though the number of targets that can fit in memory of 8KB is less (about 5%). The drop in performance for MINI from result selectivity 1% to 10% is because MINI's counting scan requires two scans for result selectivity of 10%, as it cannot store all the result targets at a time in memory. SMA is not affected by the increase in the number of targets, but performance of HHA further degrades as the number of targets are far more than the number of targets that can fit in constrained memory.

5.3 Low skew, Moderate number of targets

Figure 6(c) corresponds to Dataset D_7, wherein the data has low skew and moderate number of targets. Unlike, high skew data where few high ranked heavy targets occupy most of the database, low skew data has more moderate heavy targets. Hence, MINI takes more number of iterations to prune these heavy targets. But note that the performance of MINI is within a factor of two of SMA's and HHA's performance.

5.4 Low skew, High number of targets

Figure 6(d) corresponds to Dataset D_8, wherein the data has low skew and high number of targets. Similar to the Dataset D_7, MINI is much within a factor of two of SMA's and HHA's performance. The drop in performance from result selectivity of 1% to 10% is because the counting scan of MINI takes two scans, as the whole of result set does not fit in memory.

An important point to note in the above experiments is, although MINI's memory constraint has been **reduced by a factor of 2000** compared to SMA's memory, MINI's performance is better for high skew data and

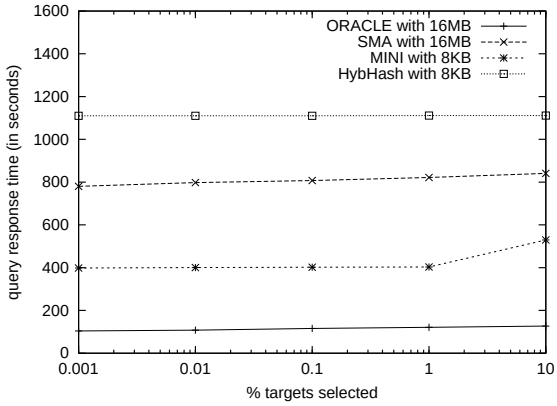

Figure 6(a): **High skew/moderate number of targets**

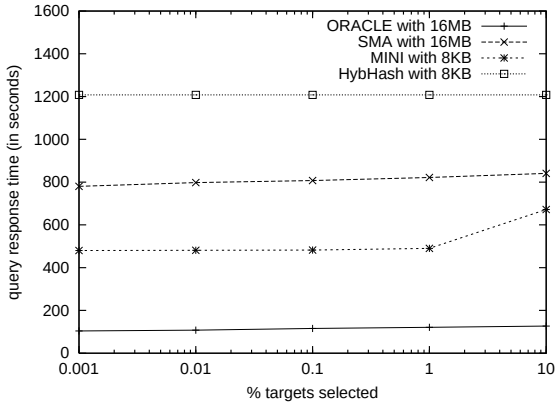

Figure 6(b): **High skew/high number of targets**

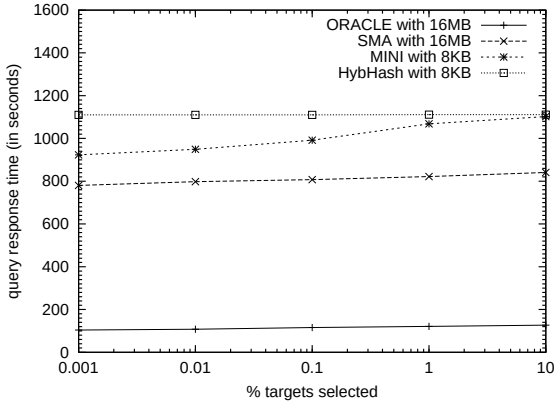

Figure 6(c): **Low skew/moderate number of targets**

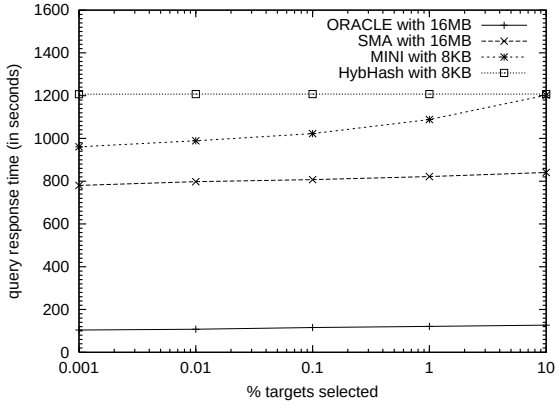

Figure 6(d): **Low skew/high number of targets**

is always within a factor of two of SMA's and HHA's performance for low skew data.

6 Recipe Algorithm

In this section, we describe a simple "recipe" algorithm (Figure 7) 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 and Low Iceberg query, that is, whether to choose CIQE, MINI, 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.

```
Iceberg_Query_Optimizer_Module (B, G, J, A, T, O, 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,
    O - comparison operator on the threshold,
    M - memory for computing the query
Output:
    C - choice of algorithm to use for computing the Iceberg Query
        CIQE, MINI, SMA.
1. if (A = AVERAGE or MIN or MAX) // irrespective of whether O is '<' or '>'
2.     return SMA
3. if (O is '>')
4.     if (A = COUNT or SUM)
5.         if (|B| = 1) // single relation
6.             b = number of hash buckets for CIQE in the available memory M
7.             if (A = COUNT)
8.                 total = N
9.             else
10.                Sample B
11.                Estimate total = aggregate value treating the whole database
                    as a single target = N/s * total_sample
12.                if (total/b < T) // takes care of average per bucket occupancy,
                    skew and selectivity
13.                    return CIQE
14.                else
15.                    return SMA
16.        else if (|B| > 1) // join of multiple relations
17.            if (J ∩ G = φ) // "no" interesting join order possible
18.                D = the amount of free disk space
19.                Estimate S = the size of the join
20.                if (S < 2 * D)
21.                    // same as |B| = 1 above
22.                else
23.                    return CIQE
24. if (O is '<')
25.     return MINI
```

Figure 7: **Recipe Algorithm**

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 3: **Crossover point : actual vs estimated**

We verified the accuracy of this binary decision for the datasets involved in our study. Table 3 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.

For Low-Iceberg queries involving SUM or COUNT as aggregate measures, MINI is the best choice (while this looks contrary to the results projected in Figures 6(a)-6(d), note that MINI's memory constraint has been **reduced by a factor of 2000** compared to other algorithms). Also for Low-Iceberg queries involving these aggregates on a join of relations, MINI is the only choice when the relations cannot be materialized although MINI works better for other cases with interesting join orders as well.

7 Related Work

Apart from the CIQE set of algorithms [1] 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 [9, 8, 13, 19, 18, 10]. In [16], 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 [17]. 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.

There has been quite some work in the area of Iceberg *cubes* recently, where the goal is to compute a restricted part of the whole data cube in order to reduce the resources required to compute and store the cube. The techniques involved here are different from that of the basic iceberg query, being primarily related to pruning the lattice that has to be computed. A bottom-up approach to computing the iceberg cube using the Apriori technique [14] for pruning is proposed in [11]. This pruning strategy is extended to handle complex measures, including averages, in [12]. Using PC clusters for parallelizing the computation of the iceberg-cube is investigated in [15].

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 we consider in this paper.

8 Conclusions and Future Work

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 a dataset with low to moderate number of targets and high to moderate skew. It never performs better than `SMA` for datasets with low skew and high number of targets.
- Performance of `CIQE` is never more than twice better than that of `SMA` for the cases where the relation is materialized and there is enough disk space to sort the relation on disk.

We defined for the first time Low-Iceberg queries, a class of queries that are similar to High-Iceberg queries, but much harder to compute. We provide a customized algorithm, called `MINI`, to handle Low-Iceberg queries, and our performance results show that it works better than `SMA` for high and low skew datasets. 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.

References

[1] M. Fang et al., "Computing Iceberg Queries Efficiently", *Proc. of 24th Intl. Conf. on Very Large Data Bases*, 1998.

[2] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Comput. Surv.*, 25, 2, 73–170, June 1993.

[3] Selinger et al., "Access Path Selection in a Relational Database Management System", *Proc. of ACM SIGMOD Conf.*, 1979.

[4] R. Ramakrishnan and J. Gehrke, "Database Management Systems", *McGraw-Hill Book Company*, 2000.

[5] H. Garcia-Molina, J. Ullman, and J. Widom, “Database System Implementation”, *Prentice Hall*, 2000.

[6] D. Bitton and D. Dewitt, “Duplicate Record Elimination in Large Data Files”, *ACM Trans. on Database Systems*, 8(2):255–265, 1983.

[7] 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.

[8] S. Chaudhari, G. Das and V. Narasayya, “A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries”, *Microsoft Technical Report - MSR-TR-2001-37*, 2001.

[9] Y. Matias and E. Segal, “Approximate iceberg queries”, *Technical report, Department of Computer Science, Tel Aviv University, Tel Aviv, Israel*, 1999.

[10] “AQUA Project”, *http://www.bell-labs.com/project/aqua/papers.html*.

[11] K. Beyer and R. Ramakrishnan, “Bottom-Up Computation of Sparse and Iceberg CUBEs”, *Proc. of ACM SIGMOD Conf.*, 1999.

[12] J. Han et al., “Efficient Computation of Iceberg Cubes with Complex Measures”, *Proc. of ACM SIGMOD Conf.*, 2000.

[13] S. Chaudhari and L. Gravano, “Evaluating Top-k Selection Queries”, *Proc. of 25th Intl. Conf. on Very Large Data Bases*, 1999.

[14] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules ”, *Proc. of 20th Intl. Conf. on Very Large Data Bases*, 1994.

[15] R. Ng, A. Wagner and Y. Yin, “Iceberg-cube Computation with PC Clusters”, *Proc. of ACM SIGMOD Conf.*, 2000.

[16] E. Segal, Y. Matias and P. Gibbons, “Online Iceberg Queries”.

[17] J. Bae and S.Lee, “Partitioning Algorithms for the Computation of Average Iceberg Queries”, *DAWAK*, 2000.

[18] I. Lazaridis and S. Mehrotra, “Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure”, *Proc. of ACM SIGMOD Conf.*, 2001.

[19] A. Gilbert et al., “Surfing wavelets on streams: one-pass summaries for approximate aggregate queries,”, *Proc. of 27th Intl. Conf. on Very Large Data Bases*, 2001.

[20] http://linuxperf.nl.linux.org/general/kerneltuning.html