Efficiently Evaluating N-Iceberg Queries

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Department of Computer Science and Automation

by

Leela Krishna Poola

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

MARCH 2002

Acknowledgements

My sincere gratitude to my guide, whose work exemplifies a *perennial enthusiasm.* I thank all my lab members for their assistance at every stage of the project, Bandi for his loving catchword *Mitrama*, Kanth and Pankaj for our *never-ending discussions*, may not be always useful, until Nagesh comes into the scene as a *moderator.* It was really amazing to say that on the final day of submission of this work for a conference, all the lab was busy to make the submission better. Believe it or not, this **chota** family inaugurated two new rooms DSL Annexe and DSL Lounge, the former is our new lab and the later for sleeping, as we used to sleep in the lab irrespective of day and night, as some were *day-batsman* and some *night-watchman.* I also thank all of my lab members for their valuable guidance and reviews which made the project a better one. I also thank my roommate Chinnu for making my stay at IISc a memorable one. Last but not the least I thank my mom, dad and sis for their moral assistance.

Abstract

An important class of queries which has recently received increasing attention is the class of queries that find aggregate values over an attribute (or set of attributes) *above* some specified threshold. This class of queries was denoted by Fang *et al.* as *iceberg queries*, because the number of tuples satisfying the threshold are very less compared to the large amount of input data. In this report we introduce the problem of answering queries which compute aggregate functions over a set of attributes *below* some specified threshold. As it finds aggregate values over an attribute (or set of attributes) *below* some specified threshold and the number of tuples satisfying the query is very less compared to the size of the database, we coin the term **N-Iceberg** queries for such type of queries. We propose an algorithm to evaluate **N-Iceberg** queries and compare them with ORACLE and traditional sorting algorithms, with very little main memory. Finally, an experimental study with various data sets is performed, where our algorithm outperforms sorting.

0.1 Introduction

Large number of applications encounter complex queries, which take long time to execute. There has been a trend in recent years to find aggregation values over an attribute (or set of attributes) *above* some specified threshold [1] [2] [3] [4] [5]. This class of queries was denoted by Fang *et al.* as Iceberg queries. They are so called because the number of *above* threshold results is often very small (the tip of an iceberg), relative to the large amount of input data (the iceberg). Iceberg queries occur frequently in data warehousing, information-retrieval, market basket analysis and clustering.

In this report, we consider the complimentary problem i.e., to find aggregation values over an attribute (or set of attributes) *below* some specified threshold. We call these type of queries as N-Iceberg queries, which stands for *Negative ICEBERG* queries. A N-Iceberg query computes aggregate function over a set of attributes *below* some specified threshold eliminating all the aggregate values above the threshold. To the best of our knowledge, we are unaware of any work that addresses the issue of computing N-Iceberg queries. In this report, we discuss the difficulties of evaluating N-Iceberg queries and describe an algorithm which efficiently evaluates N-Iceberg queries using compact main memory structures.

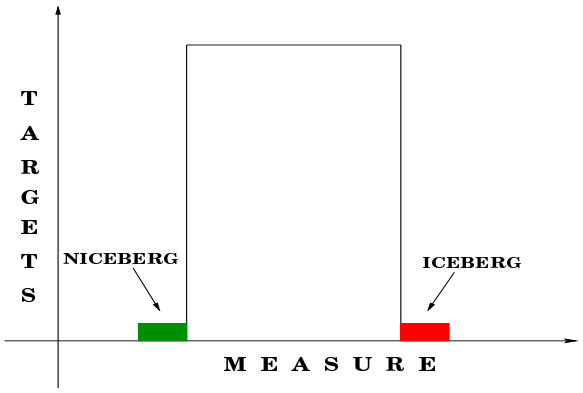

Figure 1: **N-Iceberg Vs Iceberg Queries**

Figure 1 specifies where N-Iceberg queries stand with respect to *Iceberg* queries, where "Measure" represents the aggregate value for a particular Target and "Targets" refer to the number of targets satisfying a particular Measure. From Figure 1, it is clear that the

number of targets satisfying N-Iceberg and Iceberg queries are small relative to large size of the database. Although Iceberg queries has been studied in depth, N-Iceberg queries are not addressed till now. An example N-Iceberg query, based on a relation Purchase(P.cust, P.item, P.qty), is as follows:

cust	**item**	**qty**
Har	Pep	4
Dev	Cok	5
Lee	Mar	3
Lee	Pep	8
Har	Pep	5
Dev	Cok	6
Lee	Mar	2

Table 1: **Example Database Fragment**

```
SELECT P.cust, P.item, SUM(P.qty)
FROM purchase P
GROUP BY P.cust, P.item
HAVING SUM(P.qty) <= 6
```

If the above query is executed on the database fragment shown in table 1, it results in only one tuple <Lee,Mar,2>. We refer such queries as N-Iceberg queries, as the result is very small compared to the relation P.

N-Iceberg queries have a number of useful applications like information retrieval, financial applications, marketing. For instance, a mutual fund manager may be interested in stocks that have not performed well with other stocks in the market over an extended period of time, so that he can sell them. Alternatively, consider a pharmaceutical industry, where a manager will be interested to find the pharmaceuticals which are not having good market, to discontinue producing them or produce a different kind. More examples of N-Iceberg queries are discussed in the next subsection.

The simplest way of computing a N-Iceberg query is to scan the whole database and

increment the counter for the appropriate target maintained main memory. This solution needs a distinct counter for each distinct target in the database. Since this solution uses physical main memory to store the array of counters, it will incur a performance penalty when the number of distinct targets in the data set is very large to fit in the main memory (which is always the case), in which case it is difficult to store a distinct counter for each distinct target. Another approach to answer an N-Iceberg query is to apply hashing or sorting to compute the value of the aggregate function and then remove those for which the aggregate value is more than the specified threshold. The number of tuples satisfying the condition of N-Iceberg queries is very small with respect to the total number of tuples processed, leaving room for improvements in efficiency.

We explain two different variants of sorting to compute N-Iceberg queries in the next Section.

0.1.1 What is interesting about N-Iceberg queries?

We now explain the interesting features of N-Iceberg queries and illustrate them using a few examples.

Example 1: Drowsy Student Query Consider the IISc Student database with attributes Student name, Student ID , Course ID and number of Credits for the course. Majority of the students have their aggregated credits in between 12 and 24. Students who have an aggregate sum of less than 12 credits is very small compared to the total student database. Now consider the following query, which computes all the students who have done less than 12 credits aggregated on all the courses taken by them.

```
SELECT S.ID, S.Course, SUM(S.Cred)
FROM Student S
GROUP BY S.ID, S.Course
```

```
HAVING sum(S.Credits) < 12
```

The above query is a N-Iceberg query because it results in only few targets compared to the whole student database. If current techniques like sorting or hashing are applied to compute the query, it first sorts or hashes all the distinct targets (all distinct students) and then does an aggregation of the resulting sorted or hashed database. Irrespective of the size of the result, sorting or hashing does all the work to evaluate the query.

There are many more applications in which N-Iceberg queries arise. For example, web search engines will be interested to find whether there are any documents which are very less visited, to optimize the construction of indexes on that documents.

From the above illustrated examples, the importance of N-Iceberg queries in various applications is shown. By this motivation, a solution is proposed to efficiently evaluate N-Iceberg queries.

0.1.2 Why is it difficult?

[1] presents novel and efficient techniques to evaluate iceberg queries using very little main memory and significantly fewer passes over the data, when compared to sorting or hashing. One might think that it is not necessary to formulate N-Iceberg queries. The solution for N-Iceberg queries can be inferred by just negating the solution of the Iceberg queries. However, this is not the case with N-Iceberg queries.
Intuitively, algorithms for evaluating Iceberg queries involve more of *false positives* (light targets reported as heavy, where light targets mean targets whose aggregated value is less than the threshold and heavy targets mean whose aggregated value is more than the threshold). If we consider C as the potentially heavy candidates output by the above algorithms, and F as answer to the query. Then, if $C - F$ is non empty, then the algorithm reports *false positives*. Alternatively, if $F - C$ is non empty, then the algorithm reports *false negatives* (light targets are missed). If the techniques used for evaluating

Iceberg queries are used for evaluating N-Iceberg queries, they result in *false negatives*. Eliminating *false positives* from the candidate set(C) involves post-processing that scans the whole database once and explicitly counts the frequency of all targets in the candidate set. But, post-processing to eliminate *false negatives* from the candidate set(C) is very inefficient and in fact be as bad as the original problem [1]. In other words, computing N-Iceberg queries is not easy compared to computing Iceberg queries.

There is no known efficient technique to identify the lowest frequencies in a distribution that have relatively very high frequencies and a few small ones [11]. The common techniques used to identify such lowest frequencies are sorting and hashing. In this report, we provide a novel efficient technique to address this problem. However, we are unaware of any work that tries to identify such small frequencies in a distribution.

The main contributions of this report are:

We define N-Iceberg queries and identify the usefulness of these class of queries in real world applications.

We propose an algorithm for evaluating N-Iceberg queries efficiently. We use a multi pass, partition based algorithm which, in each pass over the data, generates a mini database, which is compressed not only vertically but also horizontally. This is a write-also algorithm, where the mini database generated at each pass becomes the input to the next pass.

We evaluate our algorithms over a wide range of distributions and for different characteristics of data sets. We show that our algorithm perform better than n-way external mergesort.

0.1.3 Organization

The rest of the report is organized as follows. In Section 0.2, a suite of algorithms to evaluate N-Iceberg queries are presented. In Section 0.3, we propose the performance methodology and experimental results of our techniques compared with the traditional

sorting and ORACLE algorithms. In Section 0.4, we conclude the report along with some directions for future research.

0.2 N-Iceberg Algorithms

In this section, we explain some variants of sorting to evaluate N-Iceberg queries. We also propose a multiple-pass partition-based algorithm, `MINI`, for evaluating N-Iceberg queries.

0.2.1 The `SMA` Algorithm

In the **S**ort-**M**erge-**A**ggregate algorithm, the relation I is sorted on the target attribute using `Two-Phase Multi-way Merge-Sort` [9] (also known as `external merge-sort`). It consists of the following phases:

sort : Sort main-memory sized pieces of the data, such that every record is part of a sorted list that just fits in the available main-memory.

merge : Iteratively merge the smaller sorted sublists from the previous pass into larger sorted sublists till a single sorted list results.

Next, in the `aggregate` phase, the single sorted list, which is now stored on disk, is scanned once to compute the count for the contiguous set of tuples corresponding to each target. For heavy targets whose count exceeds the threshold, the target along with its count becomes part of the query result.

Cost of the `SMA` Components

We now discuss the cost involved in executing each of the `SMA` components. Assuming B buffer pages are available in main-memory and that we need to sort a large relation with N pages:

- the `sort` phase would result in $\lceil N/B \rceil$ runs of B pages each (except for the last run, which may contain fewer pages)

- $B-1$–way merge in the `merge phase` will require $\lceil log_{B-1}\lceil N/B\rceil\rceil$ passes to merge all the runs into a single sorted list.
- the `aggregate` phase will require single scan of the fully sorted database.

  Thus the analysis of the above steps suggests that `SMA` performance is not linear in the size of data. But as shown in the analysis done in [10], for sufficient amount of main-memory, `SMA` finishes in 3-5 passes of the database for most real-world dataset sizes.

0.2.2 The `ISMA` Algorithm

`ISMA` is a simple modified version of `SMA` that is specifically optimized for handling aggregate queries. It makes use of two optimizations, Early Projection and Early Aggregation:

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.

Early Aggregation : The aggregate evaluation is pushed into the `sort` and `merge` phases, thereby reducing the size of data that has to be merged in each successive `merge` iteration of `external merge-sort`. Further, this preprocessing makes the last `aggregate` phase redundant since the aggregate values have been already computed.

We will encounter the same costs for advanced algorithms like duplicate record elimination in large data files [6] and hash based aggregations [8].

0.2.3 MINI

A partition $P \subseteq$ relation R refers to any 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 partition-based algorithm is to partition the data, and prune the targets in a partition if their count is above the specified threshold. We now explain the theorem used for pruning

the candidates.

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

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

$$(\exists\ i\ aggr(t, P_i)) \geq T \Rightarrow aggr(t, R) \geq T$$

$$i = 1,2,...,n;$$

What the theorem states is that if a target is beyond the threshold in atleast one partition P then it must always be beyond the threshold in relation R. Hence according to the theorem, we can prune those targets that are beyond the threshold in each of the partitions. On the other hand, targets below the threshold are written to disk.

At the end of the pass, we have a new database composed of all the targets that proved to be light in some partition. This database, which we refer to as mini-database, will be smaller in size, and will be fed as input to the next iteration. This iterative process continues until all the remaining candidate targets fit in memory after which we do a counting scan of the *original* database to eliminate all the false positives.

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

- During the first pass over the original database, only the *result attributes* from the tuples are written to the mini database.
- During all passes, *aggregation* of the tuples corresponding to a common target results in vertical compression of the database.

Figure 2 shows the architecture of the algorithm. We now briefly describe the steps of the algorithm below and defer the detailed algorithm to Figure 3.

1. **Hashing Scan:** The hash size is constrained by the main-memory allocated for the algorithm. For each tuple, hash the target to a main-memory collision-resolving hash table (i.e. distinct targets occupy different buckets), and update the counter at the corresponding location. This continues till the hash table becomes full which marks the end of the current virtual partition and beginning of the next partition.

2. **Pruning Scan:** From the hash remove all the targets whose counter values are less than the threshold and write these along with their counter values to the mini database. The mini database would thus consist of targets that appear light in the current partition, but are not guaranteed to be light (false positives). But the algorithm does not have false negatives, as light targets are *always* written.

3. **Filtering Scan:** For each tuple, hash on the target value and if the target does not exist in the hash table (after the final pruning scan), write the tuple to the mini database. This ensures that the tuples for the most heavy targets identified for the current iteration have been pruned and are now being filtered from the rest of the database, thus resulting in tremendous reduction in database size from the previous pass.

4. **Counting Scan:** Scan the mini database and for each target whose counter is less than threshold, scan through the original database to remove all the false positives.

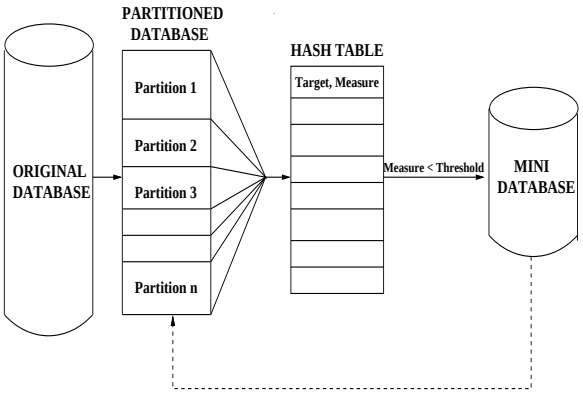

Figure 2: **Architecture of MINI**

Steps 1 and 2 are performed repeatedly as long as the pruning in step 2 results in enough hash buckets being emptied. As a result, the real heavy targets tend to accumulate in the hash table. When step 2 no longer results in pruning enough buckets, step 3 is performed next on the rest of the database for the current pass. Finally step 4 is performed when the current mini database fits in memory.

```
MINI (IP_DB, T, P)
Input : DataBase IP_DB, Threshold for N-Iceberg query T, Memory Size M
Output: Result Set F with targets less than Threshold T

1.    DB = IP_DB
2. Mini DataBase M_DB = NULL // Mini Database to store the light targets
          // Iterative calls
3. while DataBase Size |DB| > Memory Size M
4.          pruned = highvalue
5.          while (pruned > 0)
6.                    // Hashing Scan
7.                    while (HT is not full)
8.                              ReadNextTuple()
9.                              Hash on target
10.                              if (target already exists)
11.                              target.count = target.count + count
12.                              else
13.                                        insert into HT
14.                                        target.count = count
15.                   // Pruning Scan
16.                   pruned = 0
17.                   for each entry in HT
18.                             if (target.count < T)
19.                                        append to M_DB
20.                                        remove from HT
21.                                        pruned = pruned + 1
22.         // Filtering Scan
23.         while (more tuples in database)
24.                   ReadNextTuple()
25.                   Extract target and count
26.                   Hash on target
27.                   if (target does not exist)
28.                             append to M_DB
29.         DB = M_DB
30.         M_DB = NULL
31. // Counting Scan
32. for each target in DB
33.         if target.count < T
34.                   if target.count in IP_DB < T
35.                             insert target in F
```

Figure 3: **Algorithm MINI**

0.2.4 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, N-Iceberg queries result in a small set of results, it is very 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.

0.2.5 Impact of various data/query parameters

Finally, we conclude the section by discussing the impact of various data and query parameters on the performance of `MINI` and `SMA/ISMA`. We consider the following four parameters: target-count distribution, number of unique targets, size of the database and result selectivity as reflected by the threshold.

Target-count distribution

Target-count distribution will not affect SMA, as the algorithm is independent of this parameter for any particular dataset. But, it affects MINI's pruning scan which removes false positives in each pass. For high skew target-count distribution (mean much less than T), MINI performs better, because the number of heavy targets (greater then T) are more. Pruning scan will be able to remove many false positives in each pass.

For target-count distribution with low skew, where the peak target- counts are comparable to the mean for the distribution (mean comparable to T), pruning scan is not of much help, as the number of heavy targets in each pass are less. As a result, the mini

database will not reduce much. We propose a recipe algorithm in [7], to show when a query resolves to ISMA or MINI for any data distribution.

Number of unique targets

This parameter will not affect SMA much. In contrast, MINI's performance will be affected as the number of targets increase. As the number of targets increase, the number of false positives increase, and this results in mini database not reducing much. We have evaluated MINI, where it performs better than ISMA, for only a fraction of targets fitting into the constrained memory.

Size of the database

This parameter will not affect MINI much. In contrast, SMA will be affected by the increase of the size of the database. For a given number of targets, if the database size increase, then the number of targets more than T increase, thus decreasing the number of false positives, resulting in good performance for MINI. But the cost incurs in scanning the initial database.

Result Selectivity

This parameter decides the number of tuples in the result set. As the result selectivity decreases, the number of false positives increase, thus affecting MINI's performance. As the proposed solution is for iceberg query answers, where the result set is very small, this parameter will not affect much, as the result selectivity will be too high. In contrast, only the aggregate phase of SMA gets affected.

0.3 Performance Evaluation

In this section, we present the performance model and experimental result of MINI over various datasets. To depict MINI's performance, we compared it with SMA, ISMA and ORACLE.

Experiments were performed on an Intel 800 MHz Pentium III machine, running Redhat Linux 7.1. The machine has 512MBytes of memory and 36GBytes of local SCSI disk storage. The datasets considered for evaluation and the mini database generated are stored in disk. Memory is constrained to 8KBytes through out the experiments. The details of the dataset considered in our study are described in Table 2.
`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, `RS` indicates the size of a tuple (in bytes), `TS` indicates the size of the target fields (in bytes), `MS` indicates the size of the measure fields (in bytes), `Mean` represents the mean value of target counts, and `Variance` represents the variance in target counts, `Lexis Ratio` ($= Var/Mean$) is a measure of the skew in the count distribution, and `SC` represents the smallest target count.

Data-set	Cardi-nality	Unique Targets	Size of DB	RS	TS	MS	Mean	Variance	Lexis Ratio	SC
D_1	1	5000	2GB	16	4	4	25067	605080381	40159	2
D_2	1	10000	2GB	16	4	4	12516	144573262	11551	1
D_3	2	10000	2GB	16	8	4	9316	154813655	16618	1

Table 2: **Statistics of the datasets**

Different datasets are distinguished by the number of targets and the skew of the distribution. Experimental results are shown for high skew data only, as MINI will not perform better for low skew data and the better alternative is ISMA. In the following experiments moderate number of targets represent that the total number of targets re comparable to the targets that can fit in to the constrained memory. On the other hand, high number of targets means that the number of targets that can fit into the memory is very less, in our case it is 1% of the distinct targets in the dataset.

We now explain the performance graphs shown from Figure 4-6. X axis represent the query response time for various result selectivities, represented on Y axis, ranging from 0.001% to 10%.

Figure 4 corresponds to Dataset D_1, wherein the data is high skew with moderate number of unique targets. For this dataset, MINI performs better than SMA and ISMA

for a range of selectivities. For this dataset the number of targets is comparable to the number of targets that can fit in the memory.

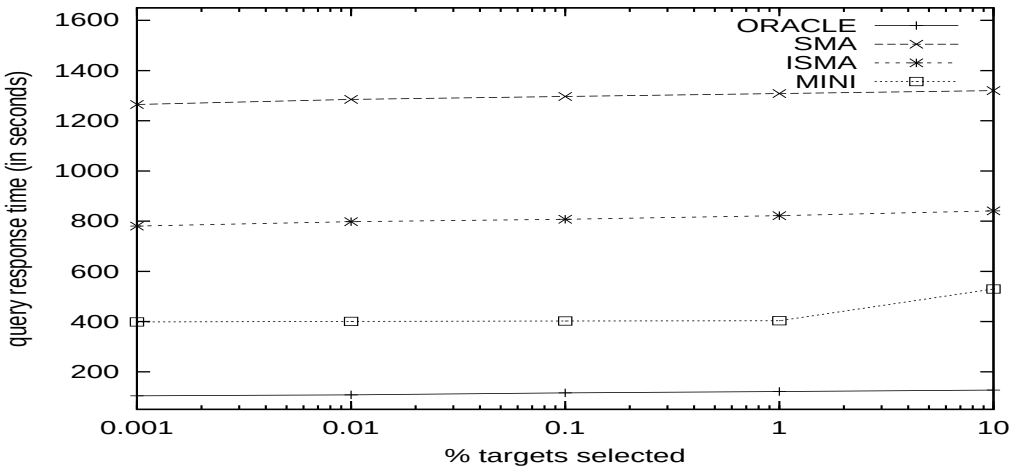

Figure 4: **high skew/moderate number of targets**

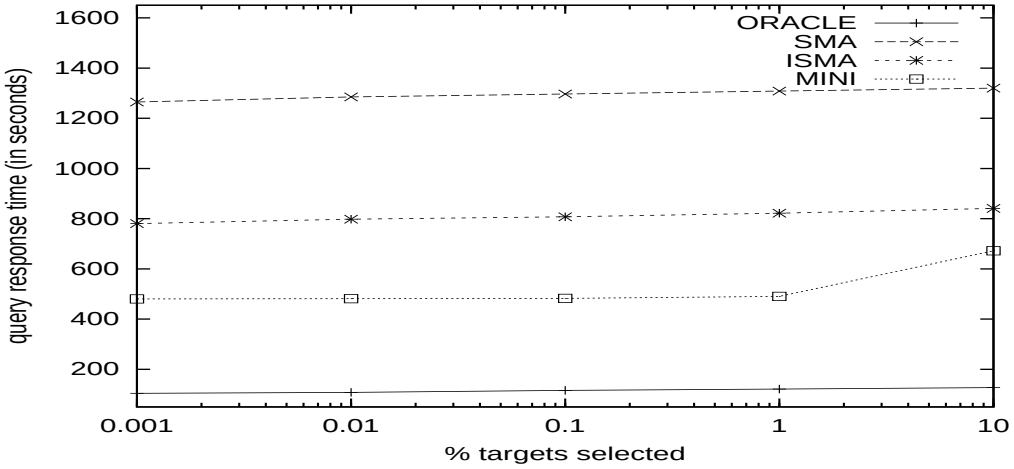

Figure 5: **high skew/high number of targets**

Figure 5 corresponds to Dataset D_2, wherein the data is high skew with high number of targets. For this dataset, MINI performs better, even though the number of targets that can fit in memory is less, in this case as the number of buckets in the memory is 1K, only 1% of the targets can fit in memory.

Figure 6 corresponds to Dataset D_3, wherein the data is high skew with high number of targets. For this dataset, MINI performs better, even though the number of targets

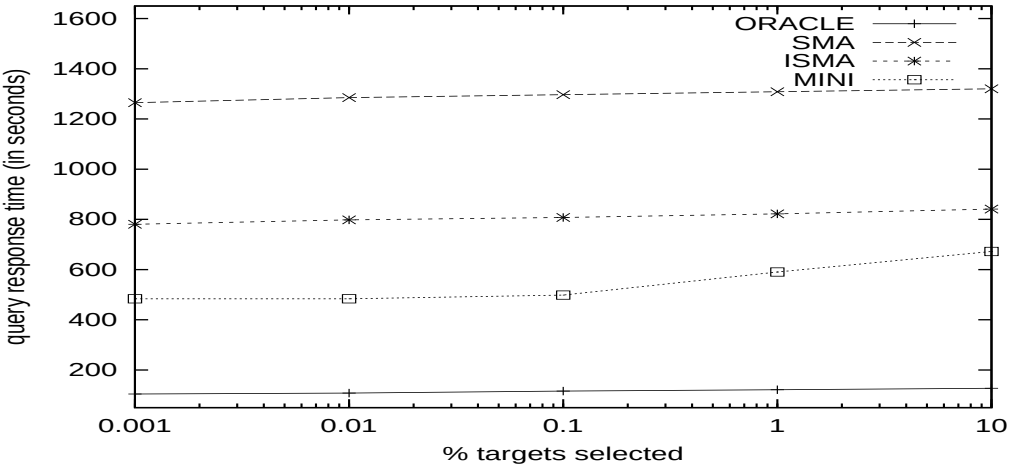

Figure 6: **high skew/high number of targets**

that can fit in memory is less, in this case as the number of buckets in the memory is 0.6K, less than 1% of the targets fit in memory.

0.4 Conclusions

We defined for the first time N-Iceberg queries, a class of queries equally important as Iceberg queries, but much harder to compute. We provide a customized algorithm, called MINI, to handle N-Iceberg queries, and put the performance of `MINI` in performance perspective against the benchmark algorithms, `SMA`, `ISMA` and `ORACLE`, and found the following:

`MINI` performs better than `ISMA` for a dataset with low/moderate number of targets with high/moderate skew. It never performs better than `ISMA` for datasets with high number of targets and low skew.

In our future work, we propose to provide a simple recipe algorithm for the incorporation of N-Iceberg queries in the Query Optimizer. This recipe takes into account the various data and query parameters for choosing between `MINI` and `ISMA`.

Bibliography

[1] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman, "Computing Iceberg Queries Efficiently", *Proc. 24th Int. Conf. Very Large Data Bases*, 1998.

[2] K. Beyer, and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg CUBEs", *Proc. 1999 International ACM SIGMOD Conference*, 1999, pp. 359-370.

[3] J. Han, J. Pei, G. Dong, and K. Wang, "Efficient Computation of Iceberg Cubes with Complex Measures", *Proc. 2001 International ACM SIGMOD Conference*, 2001, pp. 1-12.

[4] R Ng, A. Wagner and Y. Yin, "Iceberg-cube computation with PC clusters", *Proc. 2001 International ACM SIGMOD Conference*, 2001, pp. 25-34.

[5] L. Findlater and H. Hamilton, "An Empirical Comparison of Methods for Iceberg-CUBE Construction", *Technical Report CS-2000-06*, August, 2000.

[6] D. Bitton and D. Dewitt, "Duplicate Record Elimination in Large Data Files", *ACM Trans. on Database Systems*, Vol. 8, No. 2, 1983, pp. 255-265.

[7] K. Leela, P. Tolani and J. Haritsa, " On Incorporating Iceberg Queries in Query Processors", *Technical Report TR-2002-01*, DSL/SERC, Indian Institute of Science, 2002.

[8] R. Epstein, "Techniques for processing of aggregates in relational database systems", *Technical Report UCB/ERL M7918*, University of California, Berkeley, 1979.

[9] H. Garcia-Molina, J. Ullman, and J. Widom, "Database System Implementation", *Prentice Hall*, 2000.

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

[11] Y. Ioannidis and V. Poosala, "Histogram-Based Solutions to Diverse Database Estimation Problems", *IEEE Data Engineering*, Vol. 18, No. 3, 1995, pp. 10-18.