Techniques for computing Iceberg queries efficiently

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
Pankaj M. Tolani

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

MARCH 2002

Acknowledgements

The satisfaction that accompanies the successful completion of any task would be incomplete without acknowledging the people who made it possible. I wish to express my deep sense of gratitude for my project guide, Prof. Jayant R. Haritsa, for his encouragement, guidance and unstinted support throughout the project. Also many thanks to colleagues and friends for helping throughout the project. This project would not be in its present form without their support.

Contents

List of Figures

List of Tables

Abstract

Many applications compute aggregate functions over an attribute (or set of attributes) to find aggregate values above some specified threshold. These are popularly known as "iceberg queries", because the number of above threshold results is often very small (the tip of an iceberg), relative to the large amount of data (the iceberg). Traditional approach to handle these is to *sort* the relation on disk, then do a pass over it, aggregating and selecting the targets above the threshold. Recently a technique based on elimination by *scaled-sampling* and *coarse-counting* has been proposed to execute these low selectivity queries efficiently. Here we build a performance evaluation framework to evaluate the performance of these techniques for a range of system, data, and query parameters. We also compare these against `ORACLE`, a *lower-bound* optimal but practically infeasible algorithm that "magically" knows the identities of the qualifying targets without any computation. With the analysis of the practical algorithms, supported by the performance evaluation, we have designed a "recipe" algorithm, which based on parameters for a particular query, suggests which of these alternative choice of algorithms should be used for the execution of the submitted query.

Chapter 1

Introduction

Many applications compute aggregate functions over an attribute (or set of attributes) to find aggregate values above some specified threshold. These are popularly known as "iceberg queries", because the number of above threshold results is often very small (the tip of an iceberg), relative to the large amount of data (the iceberg). Such queries are common in many applications, including information-retrieval, data warehousing, market basket analysis in data mining, clustering and copy detection. A prototypical iceberg query based on a relation $I(target1, target2, :::; targetk, rest)$ and a threshold T can be written as:

```
SELECT target1, ..., targetk,
       agg_function(measure)
FROM I
GROUP BY target1, ..., targetk
HAVING agg_function(measure) > T
```

where the values of $target1, ..., targetk$ 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 relations.

STUDENT (S) relation

SRNo	Name
604100013	Leela Krishna Poola
604100110	Nagesh Bhattu Sristy
604100130	Bandi Sreekanth Reddy
604100418	Pankaj Mohanlal Tolani
604700080	Srikanth B

COURSE (C) relation

CourseID	CourseName	Credits
CD	Compiler Design	4
CG	Computer Graphics	4
DC	Distributed Computing	3
EC	ECommerce	4
EP	Entrepreneurship	4
FS	Fuzzy Sets and Systems	3
IR	Information Retrieval	3
NN	Neural Networks	3
RTS	Real Time Systems	3
TIDS	Topics in DBMS	3
TOS	Topics in OS	3

REGISTRATION (R) relation

SRNo	CourseID	Grade
604100013	CD	B
604100110	CD	B
604100130	CD	C
604100418	CD	A
604700080	CD	B
604100110	CG	C
604100013	DC	B
604100130	DC	B
604100418	EC	A
604100418	EP	S
604100013	FS	B
604100110	FS	B
604100130	IR	B
604700080	NN	C
604700080	RTS	A
604100013	TIDS	A
604100110	TIDS	C
604100130	TIDS	B
604100418	TIDS	S
604700080	TIDS	D
604100418	TOS	A
604100013	TOS	A
604700080	TOS	C

Figure 1.1: **Students' Database**

Consider the example database in Figure 1 that maintains the details of the students registering courses at the institute. Now the "Dean's Query" asking who registered for eighteen or more credits in the semester would result in only one tuple (`SRNo`="604100418") in the query result. The corresponding SQL query for the database schema is shown below:

```
SELECT S.SRNo, sum(Credits)
FROM STUDENT S, REGISTRATION R, COURSE C
WHERE S.SRNo=R.SRNo and R.CourseID=C.CourseID
GROUP BY S.SRNo
HAVING sum(Credits)>=18
```

In this case, as shown in the SQL query above, the equi-join of the three relations ($S \bowtie R \bowtie C$) with a projection of `SRNo` and `Credits` attributes forms the relation I of our prototypical query, `SRNo` being the "target" on which the aggregation is done and `Credits` the "measure" on which the threshold criteria of eighteen or more credits is applied. The key point to note here is that very few amongst all the students would actually qualify for the above iceberg query. This is also highlighted in Figure 1.2 which represents the histogram for a typical Students' Database.

1.1 Approaches for computing Iceberg queries

The simplest approach to store all targets and their measure values in a hash table [4] is applicable when the entire hash table fits into main memory. But here we consider large databases where targets are significantly in excess of the available memory, and hence do not discuss this approach any further. The traditional approach (hereafter referred to as `SMA`) used in most database query processing engines, to answer such queries is to *sort* I on disk, then do a pass over it, aggregating and selecting the targets above the threshold. If the available memory is small relative to the size of I, the sorting can take a few passes over the data on disk. A key point to note is that the amount of work done

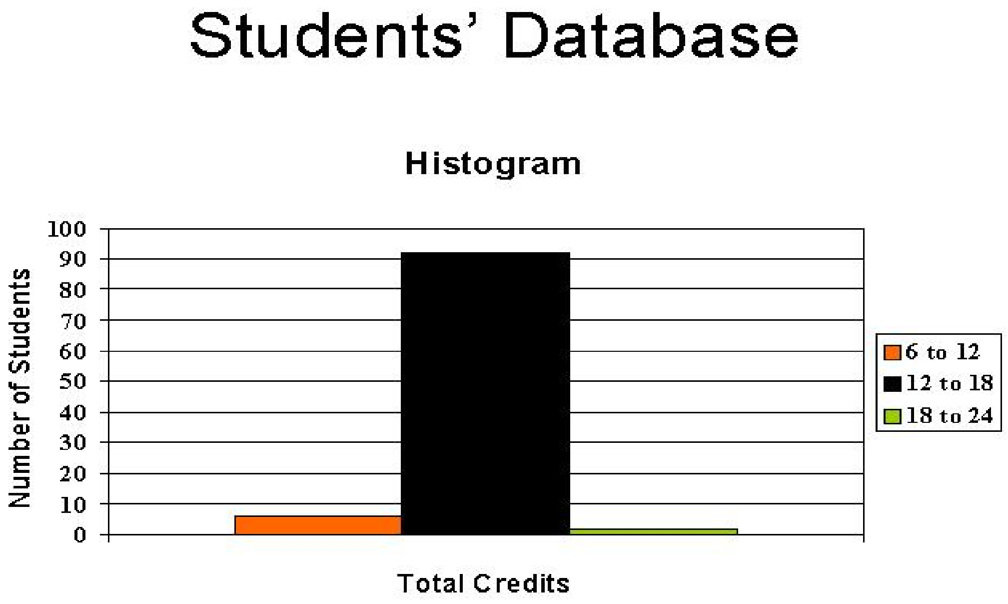

Figure 1.2: **Histogram for the Students' Database**

in query computation is *independent* of the selectivity of the query. ISMA is an optimized version of SMA and uses the early projection of result attributes and the early aggregation technique to achieve reductions in the size of the database that has to be finally sorted. Computing Iceberg Queries Efficiently [2] is a recent work that proposes a collection of algorithms (which we collectively refer to as CIQE) for the efficient computation of these queries. These use a combination of techniques like *scaled-sampling* and *coarse-counting* to eliminate the targets that *cannot* be part of the query result, and hence provide much better query response times compared to SMA as shown by the performance evaluation in [2]. The details of these algorithms are presented in Chapter 2.

1.2 Limitations of CIQE

We have identified the following problems in using the CIQE approach as opposed to SMA(and ISMA):

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

    1. The comparison predicate is $>$, [1]

    2. The aggregate operator is either COUNT or SUM,

- `CIQE` performs better than `SMA` only for low selectivities. Whether the threshold is *high* enough to ensure *low selectivity* needs to be attested either by a domain expert or by using histograms [7]. But most databases maintain only single attribute histograms, which are not sufficient to estimate accurately the selectivity for queries having target sets with multiple attributes. On the other hand, the performance of `SMA` is independent of the query selectivity.

- For a constant amount of run-time memory, the performance of `CIQE` degrades considerably as the number of distinct targets increase. This is because `CIQE` is not able to eliminate enough targets that *cannot* belong to the query result. Detailed explanation of this observation provided in Section 3.1. On the other hand, the performance of `SMA` depends mildly on the number of distinct targets.

- The performance of `CIQE` varies a lot with target count distribution. In fact, our experiments show that, in comparison to `SMA`, these perform better only for data with "log distribution" (high skew) and much worse for data with "linear distribution" (less skew). We explain the terms "log distribution" and "linear distribution" later in Section 3.2. In contrast, the performance of `SMA` is independent of the data distribution.

- Multi-Query optimizers that take a query workload consisting of a series of queries will not opt for `CIQE` *as is*, because the work done across queries is not saved. One needs to investigate the potential caching of work done using `CIQE`. On the other hand, for queries that need to be executed over subsets of the same target set, a "materialized" view of the relation sorted on the attributes in the target set can be

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

maintained, thus reducing the cost of executing the individual queries using SMA. Thus, the cost of sorting once can be *amortized* [8] over the series of queries.

- If the relation I needs to be obtained using a equi-join of the normalized relations in the database, as was demonstrated in our example, then in the case where the target set shares a few attributes with those involved in the join, *sort-merge join* could be used to reduce the work required to be done by SMA.

1.3 Contributions

First, we place CIQE's performance in perspective by (empirically) comparing it against three benchmark algorithms: SMA, ISMA, and ORACLE over a variety of datasets and iceberg queries. SMA represents the classical approach described above, and provides a viability bound with regard to the minimal performance expected from CIQE. ISMA, again described above, is a simple modified version of SMA that is specifically optimized for handling aggregate queries. 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 target count of the measure for each of these qualifying groups. [2] Note that this aggregation is the *minimum work* that needs to be done by *any* practical iceberg query algorithm, and therefore the performance of ORACLE represents a lower bound.

Second, we provide a simple "recipe" algorithm that can be implemented in a query optimizer to enable it to make a decision about the appropriate algorithmic choice to be made for an iceberg query, that is, whether to choose CIQE, ISMA or SMA. The recipe algorithm takes into account both the query characteristics and the underlying database characteristics. A novel feature of this recipe is that it takes into account the interplay of the attributes involved in the equi-join and the group-by clause, and identifies the scenarios where ISMA or SMA will win over CIQE.

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

1.4 Organization

The remainder of this report is organized as follows: we describe `SMA`, `ISMA`, `CIQE` and `ORACLE` in detail in Chapter 2. We study the performance implications of various data, query, and system parameters in Chapter 3, supported by the results of the performance evaluation in Chapter 4. Next the "recipe" is described in Chapter 5. In Chapter 6, we brief the related work, and finally, in Chapter 7, we summarize the conclusions of our study and outline future avenues to explore.

Chapter 2

Approaches for computing Iceberg queries

We describe, in this chapter, the suite of algorithms – `SMA`, `ISMA`, `CIQE` – that can be used for computing 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. Further, we will use, following [2], the term "heavy" to refer to targets that satisfy the threshold criteria, while the remaining are called "light" targets.

2.1 The SMA Algorithm

In the **S**ort-**M**erge-**A**ggregate algorithm, the relation I is sorted on the target attribute using `Two-Phase Multi-way Merge-Sort` [5] (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 targets whose count exceeds the threshold, the target along with its count becomes part of the query result.

Assuming that B buffer pages are available in memory and that we need to sort a large relation with P pages, the `sort` and `merge` phases are described in detail next:

2.1.1 Phase 1 (`sort`)

Repeat the following process for as many times as necessary.

- Fill all available memory of size B buffer pages from the original relation to be sorted.
- Sort the records that are in memory. Typically `quicksort [9]` is used for this.
- Write the sorted records from the main-memory onto new blocks of secondary memory, forming one sorted sublist.

2.1.2 Phase 2 (`merge`)

$B - 1$ buffers are reserved, one each for reading the input runs, a block at a time. The remaining buffer is reserved for writing the output run, a block at a time. Initially, the first records from the $B - 1$ runs being merged are fetched into the input buffers, and the output buffer is empty. The merging of sorted-sublists is done as follows:

- Find the smallest key among the first remaining elements of all the lists.
- Move the smallest element to the first available position of the output block.
- If the output block is full, write it to disk and reinitialize the same buffer in main-memory to hold the next output block.
- If the block from which the smallest element was just taken is now exhausted of records, read the next block from the same sorted sublist into the same buffer that

was used for the block just exhausted. If no blocks remain, leave its buffer empty and do not consider elements from that list in any further competition for the smallest remaining elements.

2.1.3 Cost of the `SMA` Components

- This `sort` phase would result in $\lceil P/B \rceil$ runs of B pages each (except for the last run, which may contain fewer pages). Each of these is sorted in main-memory using `quicksort` and written to disk.
- The $B-1$–way `merge` phase will require $\lceil log_{B-1} \lceil P/B \rceil \rceil$ passes to merge all the runs into a single sorted list.
- the `aggregate` phase will require single scan of the fully sorted database.

As per the cost for `sort` and `merge` components mentioned above, the performance of `SMA` will not scale linearly with the size of data. But for most real-world datasets, `SMA` can be completed in 3 to 4 passes, which translates into 6 to 8 reads and writes of the complete relation. For example, 4 passes would be required to sort a relation occupying 1 giga pages using 257 buffer pages, as shown in the analysis done in [4].

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.

2.3 The CIQE Algorithms

We now describe the `CIQE` algorithms and then discuss the performance of `CIQE` algorithms in general. If 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 [2] 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, the counter at $A[h(v)]$ is incremented. After completing this hashing scan of I, a bitmap array $B[1..m]$ is computed by scanning through the array A and setting $B[k]$ to one if $A[k] > T$. This step removes all false negatives, but might introduce some more false positives.

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

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

Among the `CIQE` algorithms, we have implemented `Defer-Count` and `Multi-Stage`, which were the recommended algorithms in [2] based on their performance evaluation. A brief-description of these 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`.

2.3.3 Cost of the `CIQE` Components

We now discuss the cost involved in executing each of the `CIQE` components. An important point to note is that in the case where the relation on which the iceberg query is being computed is not materialized, each scan requires *on-the-fly* `sort-merge join` of the base relations. Also, this might require initially running `external merge-sort` on the base relations to ensure that the base relations are sorted on the join attributes.

`Scaled-Sampling` is cheap, and requires random access of just a few tuples in the data. But in the case where the relation on which the iceberg query is being computed is not materialized, it is required to pick random samples from the stream resulting from the join.

`Coarse-Count` requires one data scan. For each tuple, it hashes on the target and increments the counter in the corresponding hash bucket.

`Candidate-Selection` requires a single data scan. For each target value, it figures out if the hash bucket corresponding to the target value is marked as heavy, and if so, adds it to F.

`Count` requires one data scan for the case where F fits in memory. In the case where the final F does not fit in memory, `Count` resorts to `ISMA` on a filtered database containing only tuples corresponding to the targets in F, also generated during `Candidate-Selection`.

Thus, 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.

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 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.

Chapter 3

Performance implications of various parameters

In this chapter, we present an informal discussion of the *qualitative* impact of the different data and query parameters on the relative performance of `CIQE` and `SMA/ISMA`. 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.

3.1 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, 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 `ISMA` for computing the iceberg query.

3.2 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. In contrast, 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 (log distribution), 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 distributions with low skew (linear distribution), that is, where $m \sim T$, 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 `ISMA` as its exit policy.

3.3 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. In contrast, this parameter also 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.

3.4 Summary

To summarize the above discussion, we present here a simple metric to guide us about when `CIQE` and `SMA` 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, otherwise we expect `SMA` to do better.

The following table also summarizes the discussion in this chapter. It shows, *qualitatively* in terms of *good* or *bad*, how `SMA/ISMA` and `CIQE` would perform for *high* and *low* values of these parameters.

Parameter	**Value**	**SMA/ISMA**	**CIQE**
Number of Targets	high	good	bad
	low	good	good
Target Count Skew	high	good	good
	low	good	bad
Result Selectivity	high	good	good
	low	good	bad

Table 3.1: **Impact of various query/data parameters**

Chapter 4

Performance Evaluation

Moving on from the qualitative comparison of the previous chapter, we now place `CIQE`'s performance in *quantitative* perspective by comparing it against the three benchmark algorithms: `SMA`, `ISMA`, 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 18 GB local SCSI HDD. The OS buffer-cache was flushed after every experiment to ensure that cacheing effects did not influence the relative performance numbers.

The details of the datasets considered in our study are described in Table 4.1. `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, `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 `Peak Count` represents the peak target count.

We now move on to the performance graphs for these datasets, which are shown in Figures 4.1– 4.4. 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%

Data-set	Cardi-nality	Num-Targets	Size of DB	RS	TS	MS	Mean	Vari-ance	Lexis Ratio	Peak Count
D_1	1	10M	1GB	16	4	4	6.71	11119	1657	194780
D_2	1	8.38M	1GB	16	4	4	8.00	8.20	1.27	24
D_3	2	62M	1GB	16	8	4	1.08	1675	1541	194765
D_4	2	16.4M	1GB	16	8	4	4.06	3.65	0.89	18

Table 4.1: **Statistics of the datasets**

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 [2]). 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 the `CIQE` algorithms is less than 5. Else we say the number of targets is *high*.

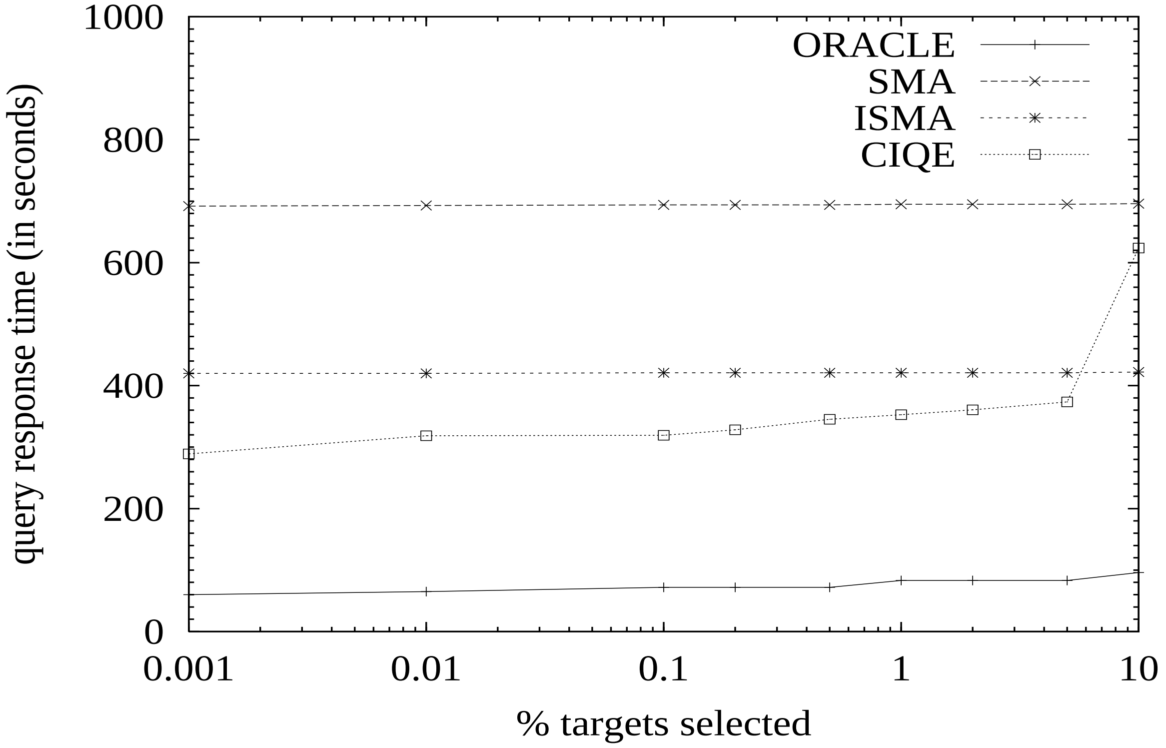

Figure 4.1: **Performance: high skew/low number of targets**

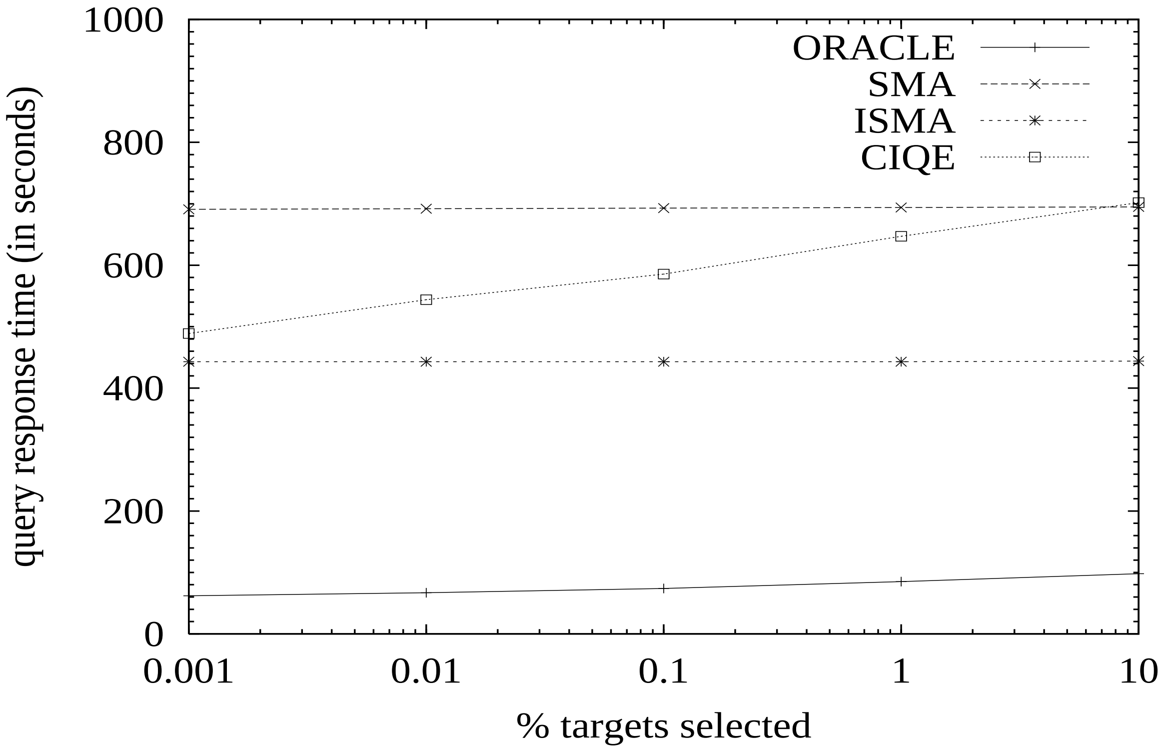

Figure 4.2: **Performance: low skew/low number of targets**

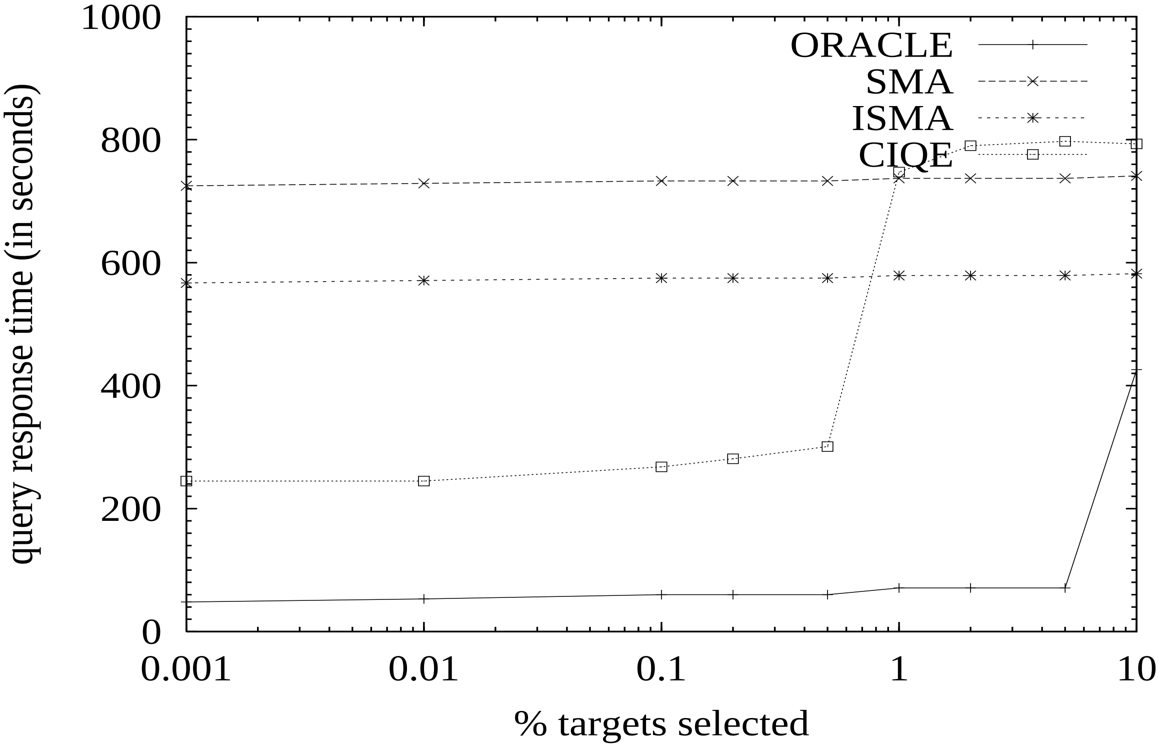

Figure 4.3: **Performance: high skew/high number of targets**

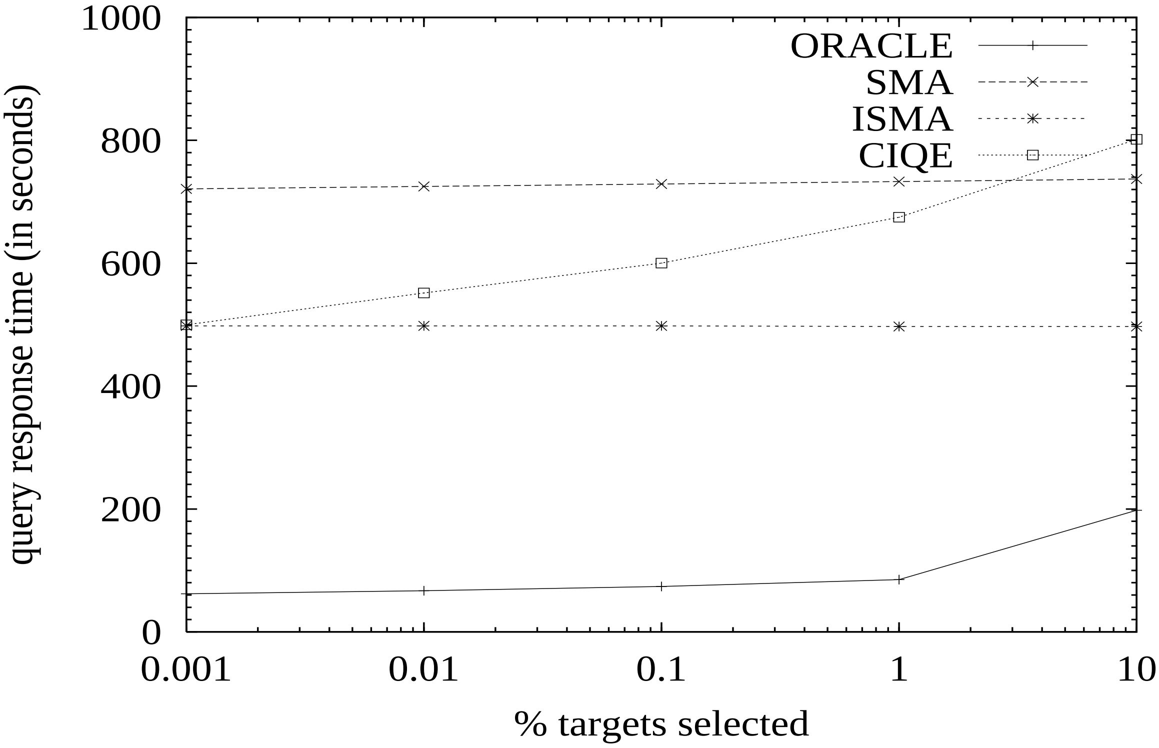

Figure 4.4: **Performance: low skew/high number of targets**

Figure 4.1 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 `ISMA` 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. Further, `ISMA` does better than `SMA` throughout the range testifying to its data reduction utility. Also note that, as discussed in Section 3.3, both `ISMA` and `SMA` are unaffected by the query selectivity, unlike `CIQE`. Finally, we see that there is a significant gap (order of magnitude) between the performance of ORACLE and the online algorithms indicating that there appears to be some scope for designing better iceberg processing algorithms.

Figure 4.2 corresponds to Dataset D_2 wherein the data now 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 `ISMA`. 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.

Figure 4.3 corresponds to Dataset D_3 wherein the data has high skew with high number of targets. For this dataset, `CIQE` performs better than `ISMA` for a much lower spread of selectivity values (only upto 0.7%). This is because the average bucket occupancy is almost 17, which is rather high. The reason that ORACLE shows a steep increase at 10% selectivity is that the result targets themselves exceed the available main memory.

Figure 4.4 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 `ISMA` because the combination of the low skew and the high bucket occupancy results in completely nullifying the pruning from the Coarse-Count step.

Another important point to note from the above experiments is that, apart from being stable across all selectivities, the performance of `ISMA` is *always* within a factor of two of `CIQE`'s performance. This means that `ISMA` is quite competitive with `CIQE`.

Chapter 5

Recipe Algorithm

In this chapter, we describe a simple "recipe" algorithm (Figure 5.1) 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 iceberg query, that is, whether to choose `CIQE`, `ISMA`, or `SMA`. A novel feature of this recipe for the query optimizer is that it takes into account the interplay of the attributes involved in the equi-join and the group-by clause, and identifies the scenarios where it would be much more efficient to use `ISMA` or `SMA` in comparison to `CIQE`.

For iceberg queries involving the `AVERAGE`, `MIN` or `MAX` aggregate functions, `ISMA` is the only choice among the suite of algorithms we consider here since `CIQE` cannot work for these functions and `SMA` always performs worse than `ISMA`.

For iceberg queries involving `COUNT` or `SUM` on a single relation, we make a binary decision between `ISMA` and `CIQE` based on the conditional in the formula on line 8. 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.

We verified the accuracy of this binary decision for the datasets involved in our study. Table 5.1 presents a summary of these results. In this table, T_{act} refers to the actual threshold (based on the experiments) below which `ISMA` starts performing better than `CIQE`, S_{act} refers to the corresponding percentage target selectivity, T_{est} refers to the

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 5.1: **Crossover point : actual vs estimated**

estimated threshold (based on the formula) below which `ISMA` 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 `ISMA` will start performing better than `CIQE` are very close to the numbers from the experimental study.

For iceberg queries involving `COUNT` or `SUM` on a join of relations, we separate the decision problem into three cases based on the intersection of the attribute sets involved in the equi-join (J) and the group-by (G). Here, we try to take advantage of the fact that if the stream resulting from the equi-join done using the efficient `sort-merge join` on attributes in J is already *partially* or *fully* sorted on the attributes in G, then computing an aggregate on the resulting stream will turn out to be a much cheaper option, without requiring any materialization of the relation. We refer to this approach as `SMJA` meaning **S**ort-**M**erge **J**oin followed by **A**ggregate. This might require sorting of the base relations if these are not already sorted on the join attributes, but sorting on the base relations will typically be much cheaper.

```
Iceberg_Query_Optimizer_Module (I, G, J, A, T, O, M)
Input:
        I - 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 - DeferCount or MultiStage
                ISMA - SMA with early projection and early aggregation
                SMJA - Sort-Merge Join followed by Aggregate

1. if (A = AVERAGE or MIN or MAX)
2.          return ISMA
3. if (A = COUNT or SUM)
4.          if (|I| = 1) // single relation
5.                  b = number of hash buckets for CIQE
                        in the available memory M
6.                  Sample I
7.                  Estimate total = aggregate value
                        treating the whole database
                        as a single target = N/s * total_sample
8.                  if (total/b < T) // takes care of mean per bucket occupancy,
                                     // skew and selectivity
9.                          return CIQE
10.                 else
11.                         return ISMA
12.         else if (|I| > 1) // join of multiple relations
13.                 if (J ∩ G = φ)
14.                         D = the amount of free disk space
15.                         Estimate S = the size of the join
16.                         if (S < 2 * D)
17.                                 // same as |I| = 1 above
18.                         else
19.                                 return CIQE
20.                 else if (J ∩ G ⊂ G)
21.                         if (grouping on attributes in G − J
                                for the individual targets based on attributes
                                in J can be done in memory)
22.                                 return SMJA
23.                         else
24.                                 return CIQE
25.                 else if (J ∩ G = G)
26.                         return SMJA
```

Figure 5.1: **Recipe Algorithm**

Case 1 - $J \cap G = \phi$ **:** In this case, a `sort-merge join` on the join attributes will not result in a stream that is even partially sorted on the attributes in G and hence the entire relation needs to be materialized for `SMA` or `ISMA` to be feasible. Also one needs temporary space equal to the size of the relation for `sort` to operate, and hence the conditional $S < 2 * D$ on line 16. Then, we use the binary decision described earlier to choose between `CIQE` and `SMA`.

Case 2 - $J \cap G \subset G$ **:** In this case, a `sort-merge join` on the join attributes will result in a stream partially sorted on the attributes in G and it might be possible to do away with the need for materialization if the tuples for the individual targets based on the attributes in J fit in main-memory. Then the approach would be to sort these in-memory target tuples on the remaining attributes in G and subsequently aggregate the counts of these tuples.

Case 3 - $J \cap G = G$ **:** In this case, it is possible to compute `sort-merge join` efficiently in such a way that the stream that results is completely sorted on the attributes in G. All that needs to be done after this is to compute the aggregate on the resulting stream.

To illustrate application of these rules, consider the sample database schema in the Introduction. Here to compute the iceberg query, `CIQE` will first sort the base relations if these are not already sorted on the join attribute. i.e. it will sort `STUDENT` and `REGISTER` relations on `SRNo` for the first join and it will sort `COURSE` and `REGISTER` relations on `CourseID` for the second join. After this it will compute the `sort-merge join` multiple times and feed the stream of tuples to the different steps in `CIQE`. But the interesting fact is that the stream of tuples is already sorted on `SRNo` and hence a simple single scan aggregate on the stream (`SMJA`) will be much more efficient.

Chapter 6

Related Work

Apart from the CIQE set of algorithms [2] 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 [12, 11, 16, 22, 21, 13]. In [19], 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 [20]. 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 (or ISMA), 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 [17] for pruning is proposed in [14]. This pruning strategy is extended to handle complex measures, including averages, in [15]. Using PC clusters for parallelizing the computation of the iceberg-cube is investigated in [18].

Chapter 7

Conclusions and Future Work

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

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

- Performance of `CIQE` is never more than twice better than that of `ISMA` for the cases where the relation is materialized and there is enough disk space to sort the relation on disk.

- For the case where the relation on which the iceberg query is being computed is obtained by a join of base relations and there is overlap between the equi-join and group-by attribute sets, `ISMA` turns to be better than `CIQE` in most cases.

We also provide 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 `CIQE` and `ISMA`. It also considers the interplay of the attributes involved in the equi-join and the group-by clause.

Bibliography

[1] P. Selinger et al., “Access Path Selection in a Relational Database Management System”, *SIGMOD*, 1979.

[2] M. Fang et al., “Computing Iceberg Queries Efficiently”, *VLDB*, 1998.

[3] M. Fang et al., “Computing Iceberg Queries Efficiently”, *Stanford DBGroup Technical Report*, 1998.

[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] T. Cormen, C. Leiserson, and R. Rivest, “Introduction to Algorithms”, *MIT Press/McGraw-Hill Book Company*, 1990.

[9] C. Hoare, “Quicksort”, *The Computer Journal*, 5(1), 10-15, 1962.

[10] F. Olken, “Random Sampling from databases”, *Ph.D. dissertation*, UC Berkeley, April 1993.

[11] 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.

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

[13] "AQUA Project", *http://www.bell-labs.com/project/aqua/papers.html.*

[14] K. Beyer and R. Ramakrishnan, "Bottom-Up Computation of Sparse and Iceberg CUBEs", *SIGMOD*, 1999.

[15] J. Han et al., "Efficient Computation of Iceberg Cubes with Complex Measures", *SIGMOD*, 2000.

[16] S. Chaudhari and L. Gravano, "Evaluating Top-k Selection Queries", *VLDB*, 1999.

[17] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules ", *VLDB*, 1994.

[18] R. Ng, A. Wagner and Y. Yin, "Iceberg-cube Computation with PC Clusters", *SIGMOD*, 2000.

[19] E. Segal, Y. Matias and P. Gibbons, "Online Iceberg Queries".

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

[21] I. Lazaridis and S. Mehrotra, "Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure", *SIGMOD*, 2001.

[22] A. Gilbert et al., "Surfing wavelets on streams: one-pass summaries for approximate aggregate queries,", *VLDB*, 2001.