

Efficient Computing of Iceberg Queries Using Quantiling

Khaled AlSabti

*Computer Science Department, College of Computer and Information Sciences,
King Saud University, Riyadh, Saudi Arabia*
alsabti@ccis.ksu.edu.sa

(Received 28 October 2003; accepted for publication 11 December 2004)

Abstract. Iceberg queries have been recently identified as important queries for many applications. These queries can be characterized by their huge input-small output. The iceberg refers to the input, and the tip of it refers to the output. We present an efficient algorithm for computing an important class of iceberg queries. This algorithm uses a focusing technique for the query result using quantiling. The new algorithm almost always requires two or less scans over the input data, which outperforms other algorithms by a factor of two or more. It has several nice properties; it scales nicely with the data size; it is robust against the data distribution. Its memory and computational requirements are small. Further, it is easy to manage. We evaluate its performance using real and synthetic datasets. We believe that the presented algorithm is the algorithm of choice for computing the queries considered in this work.

Keywords: Iceberg queries, Data mining, Large itemsets, Quantiles, Databases.

1. Introduction

With the rapid increase of the databases and data repositories sizes, new types of queries have been emerged where the output is significantly small compared to the input. Iceberg queries have been recently identified as important queries for many applications belonging to this category. These applications can be found in data mining [3, 20, 26], information retrieval [15, 18, 24, 25], decision support and data warehouse [7], web mining [9] and top k queries [10, 11]. The iceberg queries are formally introduced by Fang et al. [12]. Detailed application examples have been also presented in [12]. These queries have been extended to data cubes in [7]. Moreover, they are covered in database textbooks; e.g. [23]. These queries can be characterized by their huge input-small output. The iceberg refers to the input, and the tip of it refers to the output. Typical applications

of the iceberg queries can have very large databases; e.g. several gigabytes or more [13]. Below, we give a formal definition of the iceberg queries that we consider in this work.

Problem statement: Iceberg queries are characterized as queries with a huge input and small output. In this paper, we consider an important class of these queries, which returns frequently occurring values from a set of attributes. Below, we present a formal definition of these queries. Given a relation R that consists of n tuples each with m attributes and a set of attributes $a_{i_1}, a_{i_2}, \dots, a_{i_k}$, find the values of the tuples (i.e. the tip of the iceberg) which have attributes $a_{i_1}, a_{i_2}, \dots, a_{i_k}$, that are replicated more than a pre-specified threshold f . The assumptions are (1) relation R cannot fit into the main memory and (2) f is a relatively large percentage so that the output of the query is very small compared to the input. The type of the specified attribute(s) affects the computation requirement of the problem. For categorical attributes with a relatively small number of values, the problem is very simple and it can be solved in one scan over the data. In this paper, we assume the specified attribute(s) has a numerical type and no prior knowledge about its distribution is available. For the rest of the paper, we will refer to this query as the iceberg query.

The above problem can be found in computing frequent itemsets and association rules in data mining where the counts of the candidate itemsets are computed, followed by pruning candidates based on their counts. The survived itemsets are the tip of the iceberg [3]. Also, it can be found in web mining where documents are decomposed into chunks to find similarities among these documents [9, 12]. Further, it can be used to compute the frequently occurring words in documents to improve the performance of the information retrieval systems [15, 18, 24, 25]. Furthermore, this problem can be considered as an important primitive of knowledge discovery and data mining tasks where it can be used in early stages of the discovery process. A typical application of the above problem in databases is the *count* operator in SQL such as:

```
SELECT R.ai1, R.ai2, . . . , R.aik, count (rest)
FROM R
GROUPBY ai1, ai2, . . . , aik,
HAVING count (rest) >= n x f
```

Desired properties: The choice of an algorithm for solving the iceberg query depends on its properties. The desired properties of the iceberg query algorithms are:

1. *Number of passes:* The number of passes required over the data.
2. *Determinism:* The running time of the algorithm can be deterministic or randomized.
3. *Accuracy:* This represents the error of the output of the algorithm. Some algorithms produce exact result (i.e. zero error) and some algorithms produce approximate result with bounded or unbounded error.

4. *Sensitivity to the data order*: Some algorithms are sensitive to the order of the data and that may produce very inaccurate result for some orders.
5. *Memory requirement*: The required memory by an algorithm. Small requirements are preferable. Also we can derive a tight upper bound on this requirement for some algorithms.
6. *Management*: Ease of management for tuning the algorithm. Some algorithms are very hard to manage due to having a large number of parameters and tradeoffs.
7. *Disk space*: The extra space required by an algorithm.
8. *Materialization*: Some algorithms must work on materialized relations; which may incur additional disk space and increase the overall cost of the algorithm.
9. *Data distribution*: Some algorithms are designed for certain distributions, and they may fail for other distributions.
10. *Parallelization property*: Some algorithms are inherently highly parallelizable. In contrast, other algorithms are hard to parallelize.

Contributions: In this paper we propose an efficient algorithm for solving an important class of iceberg queries. The proposed algorithm focuses the search for the tip of the iceberg by quantiling the data based on the pre-specified threshold, followed by deriving the desired result. It has several properties; which make it the algorithm of choice for solving the iceberg query. These properties are: (1) it scales very well with the relation size, (2) it almost always requires at most two scans over the input relation, (3) it is a deterministic algorithm, (4) it generates exact result, (5) it is not sensitive to the order of the data, (6) its memory and computational requirements are very small, (7) it does not require any additional disk space, (8) it can work on non materialize relations, (9) it does not require any priori knowledge about the data distribution, (10) it is easy to manage, and (11) it is inherently highly parallelizable. We compare the new algorithm to the state-of-the-art algorithms. We conduct, also, an extensive performance evaluation of the algorithm using real and synthetic datasets and examine its behavior under different scenarios.

Paper organization: In the rest of the paper we present the related work in Section 2. We present our algorithm in Section 3. In this section, we present detailed description of the algorithm and compare it to the state-of-the-art algorithms. The performance evaluation of the new algorithm is presented in Section 4. Our conclusion is presented in Section 5.

2. Related Work

Solutions to the iceberg query can be categorized as exact or approximate solutions. In the approximate solutions, the produced result may be inaccurate. The

accuracy of these approximate solutions can be bounded or unbounded. A simple approximate technique is a random sampling approach. In this approach a small random sample is drawn from the database [22], followed by solving the problem for the small sample in-memory (e.g. using in-core sorting). This approach requires at most one pass of the data and it has a small computational requirement. It does not, however, generate exact result. Further, it can only provide probabilistic bounds on the accuracy.

Exact solutions of the iceberg query have different requirements. A simple and fast solution is based on direct addressing. In this approach an array for relation R is created and kept in memory. One scan of the data is performed to compute the frequency of each distinct value of relation R . This approach works well for a small number of distinct values of relation R . Its performance is expected, however, to significantly decrease for a large number of distinct values where we cannot keep the array in memory. Its memory requirement can be proportional to the size of relation R .

Solving the iceberg query can be done by sorting relation R , followed by scanning the sorted data to generate the result. This approach requires one pass of the sorted data plus the number of passes required by the sorting algorithm. Sorting a *large* relation requires external sorting; which is an expensive operation. It requires many read and write passes over the data as well as additional disk space. For example, a merge-based algorithm requires n/m passes, where n is the number of tuples of relation R and m is the number of tuples that can fit in the main memory. Each pass will generate a sorted run.

These sorted runs will be merged in $\log_m \frac{n}{m}$ passes. For large relations and a relatively small main memory, the external merge sort can be very expensive. One major drawback of the sorting-based approach is that it cannot directly work on non-materialized data. As pointed out by Fang et al., materializing relation R may be infeasible [12]. In this paper, they gave an example of the market basket application where materializing relation R can have quadratic increase in size over the initial size.

Computing association rules can be done using iceberg queries. Typically, mining association rules is decomposed into two subproblems: (1) generating all *large* itemsets; i.e. itemsets that have support (frequency) above a user-defined threshold, and (2) generating all rules from the large itemsets [2, 3, 20, 26]. The large itemsets can be found using an iceberg query. These large itemsets are computed in accordance to the *lattice* property of the large itemsets [3]. Large itemsets of smaller sizes are generated before longer ones using a candidates set. This set is computed using smaller large itemsets. A scan over the data is performed to compute the frequencies of the candidate itemsets. This technique requires one scan over the data plus the cost of generating the candidates set. For large candidates set, this technique may require multiple scans over the data.

Fang et al. proposed a number of approaches for solving the iceberg query [12]. These approaches, the state-of-the-art approaches, employ multiple hashing and sampling techniques to generate a candidates set of the exact result (i.e. the tip of the iceberg). This candidates set is not defined explicitly; its member must satisfy some criteria. Some of the proposed approaches generate candidates set that is a super set of the exact result; which may contain *false positives*, i.e. an item that is not part of the result. These false positives are pruned out by a post processing phase that counts the exact frequencies of these items. The cost of the post processing phase is a function of the candidates set size; it may require multiple scans of the data. Other proposed approaches may generate candidates set that is not a super set of the exact result; which may lead to false positives and *false negatives* errors. The false negatives are items that are part of the result, but are not contained in the candidates set. Detecting and finding false negatives are hard problem; it is basically the initial problem; i.e. solving the iceberg query. The proposed algorithms can work directly on non-materialized relations. Further, they can solve iceberg queries with different types of aggregates; e.g. *count* and *sum*. However, these approaches can be sensitive to the data skew. Setting reasonable values to the many parameters of the algorithms (e.g. sample size, number of hash functions, hash functions and memory management parameters) can be a non-trivial task. Further, the proposed algorithms require at least few passes over the data to compute the iceberg query. The most efficient algorithms (e.g. DEFER-COUNT, MULTI-STAGE, MULTISCAN-SHARED2) require at least four scans. Finding an appropriate algorithm for a given input can be a difficult task as it is shown in the "rule of thumb" section [12].

Quantiling: The ϕ -quantile of an ordered sequence of data values is the element with rank $\phi \times n$, where n is the total number of values. The median of a set of data is the 0.5 quantile. Quantiles are important order statistics. Their accurate estimates are required for the solution of many practical applications. The problem of finding a ϕ -quantile of a set of elements of size n which reside in the main memory can be solved in $O(n)$ time by using the deterministic algorithm of [8] or in $O(n)$ expected time by using the randomized algorithm of [14]. Large applications require an effective quantiling algorithm that reduces I/O requirements as much as possible without significantly sacrificing the accuracy. In many cases the exact value of the quantile is not needed and a good estimate of the true value is sufficient. Several approaches have been proposed in the literature; e.g. [4, 5, 16, 17, 19, 21].

3. Our Approach: Quantile-based Approach

In this section we present an efficient new algorithm for computing an important class of the iceberg queries. The new algorithm has many desirable characteristics that make

it the algorithm of choice for solving the iceberg query considered in this paper. We decompose the query computation task into two phases:

- **Phase I:** Generate a super set of the tip of the iceberg; i.e. the candidates set.
- **Phase II:** Filter out the false positives by scanning the data.

Similar decomposition was employed in [12]. There are, however, major differences between our approach and the approaches presented in [12]. Clearly the cost of the above approach depends on the cost of generating the candidates set and its size. Ideally, we would like to generate a *small* candidates set as much as possible in one pass over relation R . The smallest candidates set is the exact answer of the iceberg query itself; which do not have any false positives. Moreover, we would like to filter out the false positives in at most one additional pass. Thus computing the iceberg query can be done in two (or less) passes over R . The above scheme does not generate false negatives. This is a desirable property since handling those items is difficult task as shown in [12]. Without loss of generality, we assume that the number of attributes k (cf. the Introduction) that defines the iceberg query is one. The algorithms presented in this section can be easily extended for more than one attribute. For the sake of explanation, let us consider the following example. Given relation R (shown in Table 2) and threshold f of 20%, one can solve the iceberg query of this relation by sorting R , then computing the frequency of each distinct value. Fig. 1 shows relation R sorted. Clearly, the iceberg query result consists of 45 only. Alternatively, one can find the iceberg query result of this example using the above decomposition by first generating a candidates set of the query result as the set consisting of q_i elements shown in the figure. These elements are called *quantiles*. Thus, the candidates set consists of $q_{0.2}$, $q_{0.4}$, $q_{0.6}$ and $q_{0.8}$. Then the exact frequencies of these quantiles are computed to filter out the false positives. The following lemma presents the general case of the candidates set generation scheme.

Lemma 1 *Given relation R , the result of the iceberg query of R for threshold f ($\text{Iceberg}(R)_f$) satisfy the following.*

$$\text{Iceberg}(R)_f \subset \{q_i, q_{2f}, \dots, q_1\}$$

Proof: We proof this result by contradiction. Let us assume that there exists an element x

such that (1) $x \in \text{Iceberg}(R)_f$ and (2) $\neg(x \in \{q_i, q_{2f}, \dots, q_1\})$

Then $x \neq q_i \quad \forall i$

Then $q_{if} < x < q_{(i+1)f}$ for some i

Then $|\{y | y \in R \text{ and } q_{if} < y < q_{(i+1)f}\}| \leq fn$, where n is the total number of tuples.

Therefore the number of tuples with values x is less than fn . Thus x is not part of the result of the iceberg query, and that is a contradiction.

Table 1. Notation used throughout the paper

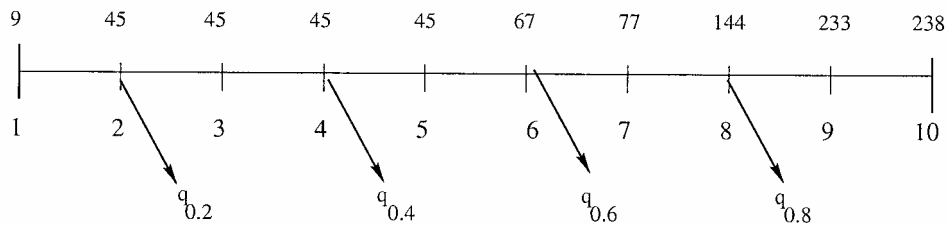
Term	Description
R	the input relation; it does not have to be materialized
n	$ R $
m	number of elements that can fit into the main memory
f	the iceberg query threshold
q_f	the f -quantile
L_{q_f}	a lower bound of q_f
U_{q_f}	an upper bound of q_f
OPAQ	the quantile algorithm [5]
s	a parameter of OPAQ

Table 2. Example of relation R

Tuple ID	1	2	3	4	5	6	7	8	9	10
Value	238	45	9	45	67	45	45	77	144	233

The candidates set can be directly generated using the above lemma. Formally, the candidates set = $\{q_f, q_{2f}, \dots, q_1\}$. The false positives among these quantiles are filtered out by scanning R . Finding the *exact* quantiles for large data that cannot fit into the main memory may require at least two passes [5]. However *accurate* estimates of quantiles can be computed in one pass over the data [5]. For each quantile, lower L and upper U bounds of the true value are generated (Fig. 2). Further, the number of distinct values in the interval defined by these bounds can be bounded [5]. Using these bounds, one can generate a candidates set as follows:

$$\text{candidates set} = \bigcup_i \{x \mid x \in R \text{ and } x \in [L_{q_{if}}, U_{q_{if}}]\} \quad (1)$$

Fig. 1. Example: relation R sorted.

It is easy to prove that this candidates set is a super set of the result of the iceberg query. This is an immediate result from Lemma 1 and the definitions of the lower and upper bounds.

Overview We solve the iceberg query using the above decomposition. In Phase 1, we generate a candidates set using equation (1). The quantile estimates can be generated using an efficient quantile algorithm for large dataset. There are a few such algorithms in the literature. We choose to use the OPAQ algorithm for this purpose [5]. This algorithm requires one pass over the data to produce the estimates. Also, it generates a relatively small candidates set. This is a desirable property because it ensures efficient implementation of the next phase. In Phase II, we determine the result of the iceberg query by filtering out false positives, if any. This phase can be done in zero or one pass over the data for candidates' sets that can fit into the main memory. Otherwise it can be staged.

Detailed Description For a given threshold f we compute accurate estimates of q_f, q_{2f}, \dots, q_1 using a quantile algorithm. The quantile algorithm must satisfy the following criteria:

1. produce lower and upper bounds of the true quantile,
2. guarantee a bounded (small) number of distinct values between the lower and upper bounds of the true values,
3. its overall cost is small and it is scalable to large databases.

The OPAQ algorithm meets all the above criteria. It requires only one pass over the data to compute the quantiles. Its computational and memory requirements are linear and sub linear in $|R|$, respectively. OPAQ generates lower and upper bounds of each quantile; i.e. L_q and U_q . Further, it bounds the number of distinct values in the intervals defined by the lower and upper bounds by $2n/s$, where s is a parameter of the OPAQ algorithm. Fig. 2 shows the output of the OPAQ algorithm. Now, we need to determine the query result from the candidates set; i.e. Phase II. There are two main steps that need to be considered. In the first step, we prune the candidates set. The frequencies of the remaining elements, if any, are computed in the second step.

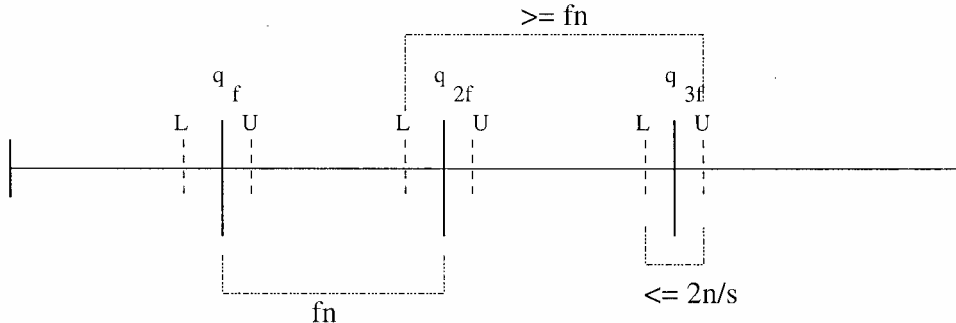


Fig. 2. True, lower and upper bounds on quantiles and some of their relationships.

- **Step 1** In this step we prune intervals based on the intervals' boundaries. These pruned intervals are not considered for further processing. For some intervals, we can prove their relationship with the query result; i.e. part or not part of the result. There are a number of cases that can be considered for performing pruning. Below we discuss four such cases.
 1. **Case 1:** For adjacent intervals with $L_{qif} = U_{q(i+1)f}$ (Fig. 2), we can determine an element of the query result from the intervals' boundaries. It can easily be shown that an element x that is equal to L_{qif} and $U_{q(i+1)f}$ for some i is part of the query result. Further, the intervals $[L_{qif}, U_{qif}]$ and $[L_{q(i+1)f}, U_{q(i+1)f}]$ consist of only one element (i.e. x) and they can be pruned out from the remaining processing.
 2. **Case 2:** For adjacent intervals with $U_{qif} = L_{q(i+2)f}$ (Fig. 2), interval $[L_{q(i+1)f}, U_{q(i+1)f}]$ is part of the query result and it can be pruned out.
 3. **Case 3:** An interval $[L_{qif}, U_{qif}]$ is not part of the query result if $U_{q(i-0.5)f} \neq L_{qif}$ and $L_{q(i-0.5)f} \neq U_{qif}$. This interval can be pruned out. Note that $U_{q(i-0.5)f}$ and $L_{q(i-0.5)f}$ are computed on the fly using OPAQ without any additional I/O.
 4. **Case 4:** An interval $[L_{qif}, U_{qif}]$ is part of the query result if $L_{q(i-0.5)f} = U_{q(i-0.5)f}$. Also, $L_{q(i-0.5)f}$ and $U_{q(i-0.5)f}$ are computed on the fly using OPAQ without any additional I/O.
- **Step 2:** For other intervals (i.e. alive intervals), we need to compute the exact frequency of all their elements to filter out the false positives. Note that in case there is no alive intervals, Phase II is completed without any additional scanning of R and computing the iceberg query took one scan of R . The size of the alive intervals (i.e. the number of distinct values) can affect the cost of Phase II. In case that all the alive intervals can fit into the main memory, finding the query result can be done by scanning R and computing the elements frequency in memory as follows. An element, first, is pruned out if it is part of the iceberg query result (Step 1) or it does not belong to any alive interval. Otherwise, it is assigned to the first alive interval such that it belongs to. Note that an element can belong to two alive intervals. Each alive interval is sorted incrementally.¹ Each element of the sorted interval consists of a value and its current frequency. In case the alive intervals cannot fit into the main memory, we need to process them in stages. We expect, however, this almost always will not be the case (see, below, the memory requirement of the alive intervals). Note that the actual counts of the output can be computed in the second phase,

¹ One can use other techniques for counting, e.g. hashing.

if needed. In what follows, we derive an upper bound of the size of the alive intervals.

The size of the alive intervals The maximum size of an alive interval is $2n/s$ (this result was derived elsewhere [5]). The maximum number of the alive intervals is $1/f$. Thus an upper bound of the requirement of processing the alive intervals is proportional to $\frac{2n}{fs}$.

We can set s to be a function of n , e.g. $s = 1\%n$. Thus the memory requirement is $200/f$. For $f = 20\%$, this bound is less than 1K and it is less than 20M for $f = 0.001\%$. In the best case, there are no alive intervals. The best case for the alive intervals is one element for each interval; the memory requirement is $1/f$. Clearly, the memory requirement of Phase II, in the worst-case, is small and it can be almost always met.

Analysis The new algorithm is very attractive. Its memory, disk, I/O and computational requirements are very small. The overall memory requirement is the maximum of the memory requirements of Phases I and II. Phase I requirement is exactly the requirement of the quantile algorithm. With OPAQ, this is $O(\sqrt{ns})$ for a given n and s . For $s = 0.1\%n$, this requirement is proportional to $1\%n$. One of the features of OPAQ is that we can control the value of s ; thus for very large relations we can use a small value of s to meet the memory limitations. The memory requirement of Phase II is proportional to $\frac{2n}{fs}$. For large relations, the memory requirement of the new algorithm is the memory requirement of the quantile algorithm; it is $O(\sqrt{ns})$ with OPAQ. The new algorithm does not require an additional disk space, even for non-materialized relations (cf. Characteristics below).

The I/O requirement of the new algorithm is the sum of the I/O requirements of Phase I and Phase II. For Phase I, OPAQ makes one scan over the relation. In Phase II, one extra scan (or less) is required in almost always all the cases. Thus we can compute the result of a query in two or less scans over relation R . The computational requirement of the new algorithm is, again, the sum of the requirements of Phase I (the quantile algorithm) and Phase II. With OPAQ, Phase I can be done in $O(n \log s)$ time [5]. Phase II requires, assuming fixed f and s is a small fraction of n , $O(n \lg A)$ (where A is the number of alive intervals) and $O(1)$ in the worst-case and best-case, respectively. Thus the overall computational requirement of the new algorithm is $O(n \log s)$. Table 3 summarizes the different requirements of the new algorithm.

Table 3. Resources' requirements of the new algorithm

<i>Resource</i>	<i>Cost</i>
Memory	$O(\sqrt{ns})$
Disk I/O	Almost always two or less scans of R
Computational	$O(n \log s)$

Limit on Threshold f Clearly, the value of f can affect the cost of solving the iceberg queries since the size of the result of an iceberg query is inversely proportional to the value of f . From the definition of the iceberg query problem (cf. the Introduction), the value of the threshold f is a percentage of n . For example with $f = 1\%n$, the upper bound of the query result size is 100. In contrast, it is 10,000 for $f = 0.01\%n$. The value of f that can be handled by our algorithm is constrained by the quantile algorithm limitations as well as those of Phase II. With OPAQ, we have the following constraints [5]:

$$\frac{1}{f} = O\left(\frac{m^2}{n}\right) \quad (2)$$

$$s \geq \frac{10}{f} \quad (3)$$

Phase II, in the worst-case, has the following constraint.

$$\frac{2n}{fs} < m \quad (4)$$

The above constraint is required so that the entire phase can be guaranteed to be completed in one scan over R . In case that it is not satisfied, this phase must be staged.

This corresponds to $\frac{200}{f} < m$, assuming that $s = 0.1\%n$. Thus, the new algorithm is very scalable with respect to the value of the threshold.

Characteristics The new algorithm has several characteristics that make it the algorithm of choice. The new algorithm requires (at most) two passes over the data for almost always all the cases independent of the data distribution. The memory requirement of the new algorithm is very small. For $s = 0.1\%n$, it is proportional to $3\%n$. The new algorithm does not require any extra disk space. This feature is important for large applications. Another feature of the algorithm is that it does not require materialized relation to produce the result. It can work directly on non-materialized relations. Again, this is important feature for large applications. Its running time is deterministic as shown in Table 3. It is very accurate; it produces exact result. It is not sensitive to the data order, i.e. its running time and result does not depend on the order of

the data. Moreover, it does not require any prior knowledge about the data distribution. The new algorithm can be efficiently parallelized. For Phase I, the OPAQ algorithm is highly parallelizable [5]. The main operations of Phase II are I/O and counting. Parallelizing I/O is straightforward, and it is optimal or near-optimal. For the counting operation, it can be performed in two stages. In the first stage, we perform a local count, which is optimal or near optimal. The second stage (i.e. the overhead) can be done efficiently using a global combine primitive [27] of size (at most) $\frac{2n}{sf}$. The cost of this operation (the overhead) is very small compared to the overall cost. We expect the parallel version of our algorithm to exhibit good scale-up, size-up and speedup characteristics.

Our Approach vs. Previous Approaches We compare our approach to the random sample approach and to those of Fang et al. [12]. The random sampling approach is simple and has small requirements compared to other approaches. For applications that require exact results or bounded non-probabilistic errors, this approach is a poor choice (cf. Section 2). The approaches of Fang et al. exhibit nice properties (cf. Section 2). However, the "best" approaches of these approaches (e.g. DEFFER-COUNT, MULTI-STAGE, MULTISCAN-SHARED2 and UNISCAN) suffer from at least one of the following:

1. The I/O requirements are large compared to our algorithm. Their requirements can be higher than those of the new algorithm by a factor of two (or more) since they require at least four I/O scans.
2. The computational requirements are linear in n . However, they use multiple hashing techniques; which can be a relatively expensive.
3. The memory management required for achieving a good performance can be a difficult task.
4. They have a relatively large number of parameters. Setting these parameters for achieving good performance can be a hard task.
5. Sensitivity to the data skew. Some of the proposed algorithms in [12] are sensitive to the data distribution.

We can conclude from the characteristics of the new algorithm and the above discussion that the new algorithm is a better option for computing the iceberg queries that we consider in this paper. This is due to (1) its small memory, computation and I/O requirements, (2) ease of management, (3) a small number of parameters, (4) its robustness to the data distribution and others (cf. Characteristics section, above).

4. Performance Evaluation

In this section we evaluate the performance of our algorithm. We have implemented the new algorithm in C and ran it on Linux Red Hat version 6.0. The machine used has the following configuration: Intel Pentium III, 512 KB cache and 128 MB RAM. The parameters of the new algorithm are n , f , data distribution and the available memory m . Below, we describe the data distributions that we used.

Datasets We want to study the behavior of the new algorithm under different known distribution. For this, we have used Uniform and Zipf [28] distributions. We also use a large real world dataset. Brief descriptions of these datasets are giving below:

1. **Uniform:** For a given size n , we generate a data where the elements are drawn randomly from $[0..n/1000)$.
2. **Zipf:** The Zipf distribution models many real life phenomena [28]. It has a parameter that determines the skew degree. This parameter p ranges between zero and one where the degree of skew is inversely proportional to its value. A value of 1 results in Uniform distribution, where a value of zero results in very skewed distribution. We generated Zip0.8, Zip0.6, Zip0.4, Zip0.2 and Zip0.0 datasets using 0.8, 0.6, 0.4, 0.2 and 0.0 as values of this parameter; respectively. For a given size n and a given p , we generate a data where the elements are drawn from $[0..n/1000)$ using Zipf distribution with p .
3. **Real:** We use a network connections dataset, which was used in the 1999 KDD intrusion detection contest (KDD-Cup99) [1]. This data has a number of attributes (continues and discrete). This dataset can be accessed in [6]. The task of the KDD-Cup99 is to build a predictive model to detect illegitimate connections called attacks. This dataset has training and testing versions. In this work, we use the first 4.5M values of the 5th and 6th attributes of the training dataset. We call these datasets Att5 and Att6. The data distributions of these columns are not known.

We performed four main experiments. For each experiment, we mainly report the total execution time, the percentage of the processed data in Phase II, the memory requirement of Phase II and number of I/O scans performed. For all these experiments we set s to $10/f$.

Sensitivity to the data size In this experiment, we set the available memory to 10 Million elements, data distribution to Zip0.2 and vary n (20 M, 40 M, 60 M, 80 M and 100 M elements) and the threshold f (0.01%, 0.05%, 0.1%, 0.5%, 1%, 5% and 10%). Fig. 3 shows the total execution time in seconds for a number of settings. We observe the following from this experiment. The algorithm made at most two I/O scans over the data. For some cases, it required one scan only; e.g. for a relatively large threshold f . This explains the sudden drop in the execution time; e.g. 100M & 1%, and 40 & 5%. Note that this depends on the effectiveness of the pruning in Phase II. Using the current

pruning technique, it is expected to get a lot of pruning for a large threshold and/or a small degree of skew in the data distribution. The second observation is the robustness of the algorithm against the threshold values for the same requirements of the I/O scans. Put other way, the overall cost of Phase II is not very sensitive to the threshold given that a second I/O scan is performed. This is because the I/O cost and a significant portion of the computation cost are independent of the threshold. The third observation is that the algorithm scales very well with the relation size; its scalability is a sub-linear. Fig. 4 shows the number of elements of the alive intervals in Phase II. For all the cases, the total size of the alive intervals is very small; it varies between 16% and 0%. As expected the total size decreases with the increase of the threshold. Note that the total size of the alive intervals is the upper bound of the memory requirement of Phase II. The exact memory requirement is the number of distinct values of the alive intervals, and it was less than 1% for all the cases.

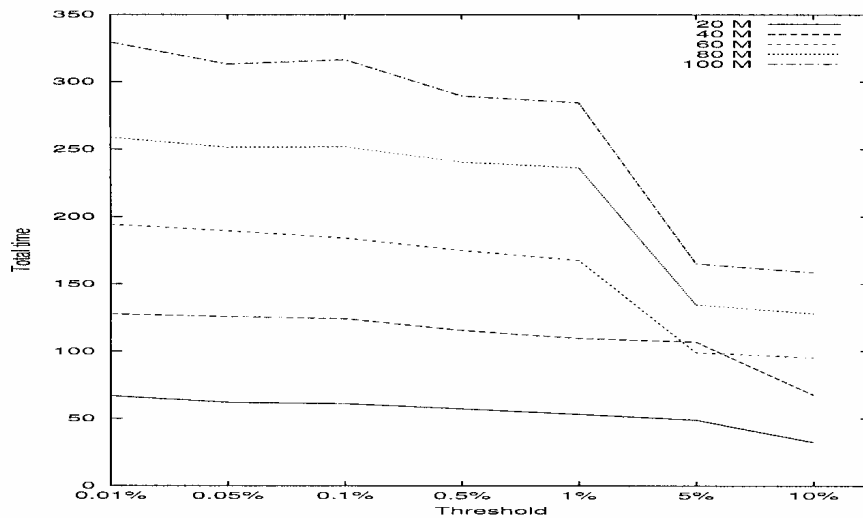


Fig. 3. Dataset size vs. threshold: Total time in seconds.

Sensitivity to the data distribution We set the data size n to 60 Million and the available memory to 10 Million elements. We vary the data distribution (Uniform, Zip0.8, Zip0.6, Zip0.4, Zip0.2 and Zip0.0) and the threshold. Fig. 5 shows the total execution time of a number of settings. We observe the following. First the total execution can be sensitive to the data distribution (equivalently the threshold). The overall cost of the algorithm can decrease by a factor of two for the same data size but different distribution (or threshold). This is because the I/O scans and the associated computation requirement can decrease by a factor of two. This explains the sudden drop of the execution time for some cases. For the Uniform case, there was no sudden drop

since its I/O requirement was one scan for all the tested threshold values. It is interesting to notice the time of the sudden drop of the execution time. It was 0.05%, 0.5%, 1%, 5% and 10%) for Zip0.8, Zip0.6, Zip0.4, Zip0.2 and Zip0.0; respectively. One can estimate

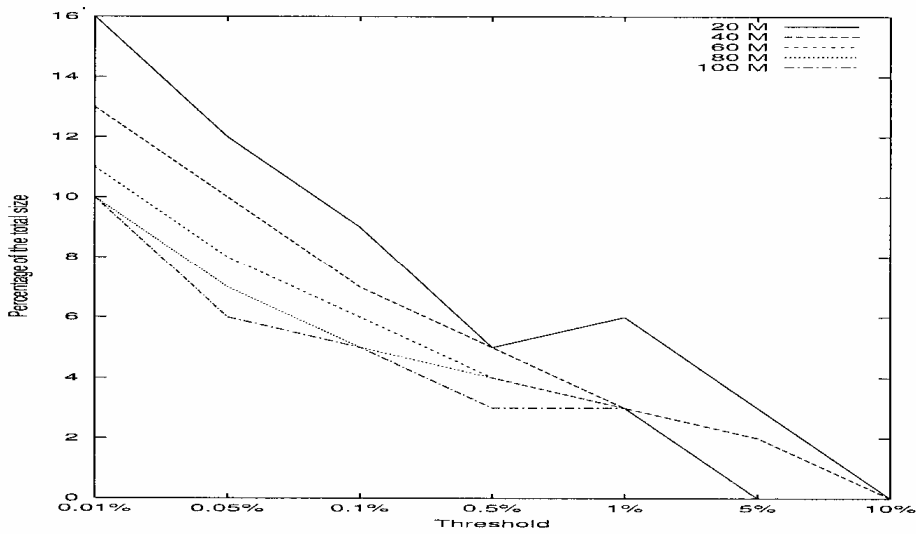


Fig. 4. Dataset size vs. threshold: Percentage of data falls in the alive intervals.

the skew degree of a dataset or compare a number of datasets by trying different threshold values. The overall cost of the algorithm for different data distributions is comparable in case it requires the same number of I/O scans. This is attributed to the fact that a little of the performed work is dependent of f ; e.g. generating the sample points and pruning the intervals. Fig. 6 shows the total number of elements of the alive intervals in Phase II. It varies between 11% and 0% of the data size. Further, it decreases with the increase of the threshold. The curve of the Uniform case is not shown since it is always zero. Again, the exact memory requirement is the number of distinct values of the alive intervals, and it was less than 1% for all the cases.

Sensitivity to the available in memory We set the data size to 60 million elements and the data distribution to Zip0.2, and vary the available memory (1 million, 5 million and 10 million) and the threshold. Fig. 7 shows the total execution time of the algorithm. Due to the small memory requirements of the new algorithm, the total execution time was virtually independent of the available memory. For all the cases, the sample points fit into the available memory and the memory requirement of Phase II was very small; i.e. less than 0.1%. It is interesting to notice that using 1 million of memory results in marginally better performance. We attribute this phenomenon to the cache effects since

we are not optimizing the memory accesses to effectively use the memory hierarchy of the machine. The total number of the alive intervals is virtually the same for the different memory sizes Fig. 8. Note the sample points can be different for different memory sizes due to behavior of the quantiling algorithm that we use.

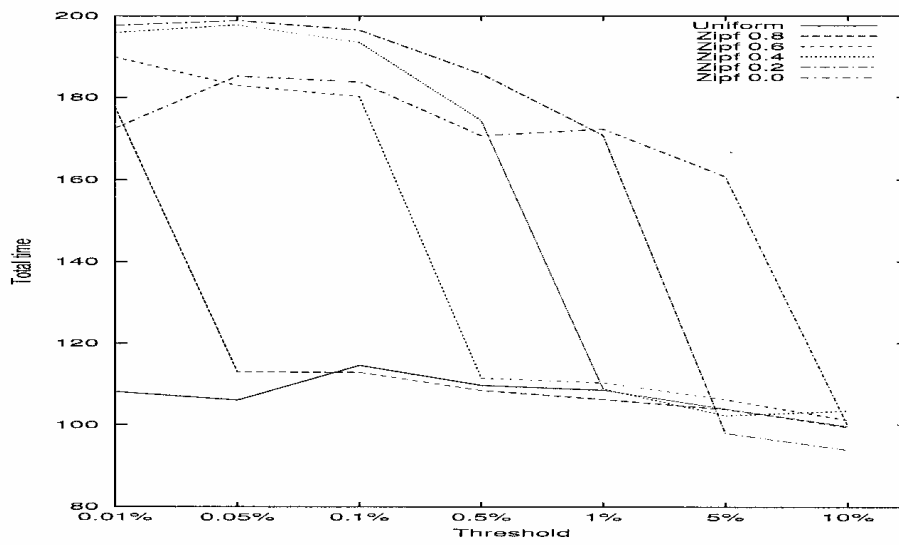


Fig. 5. Data distribution vs. threshold: Total time in seconds.

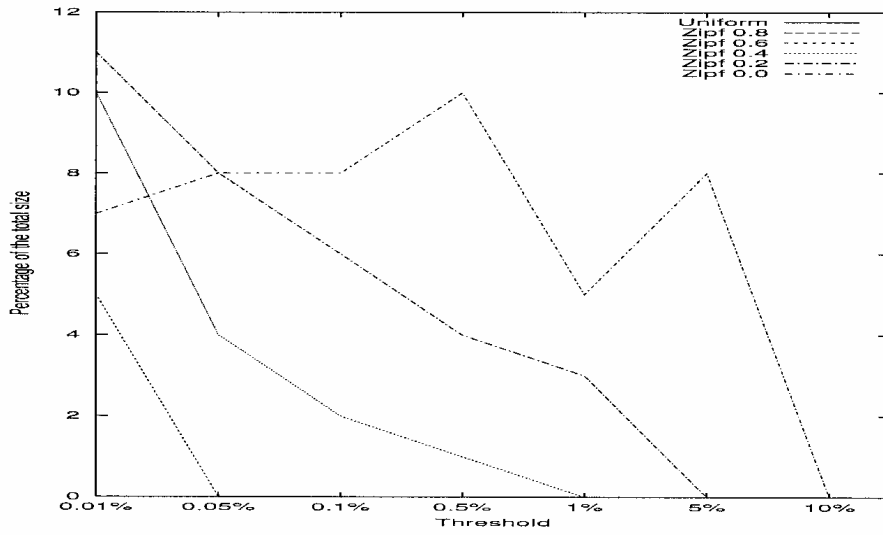


Fig. 6. Dataset distribution vs. threshold: Percentage of data processed in the second phase.

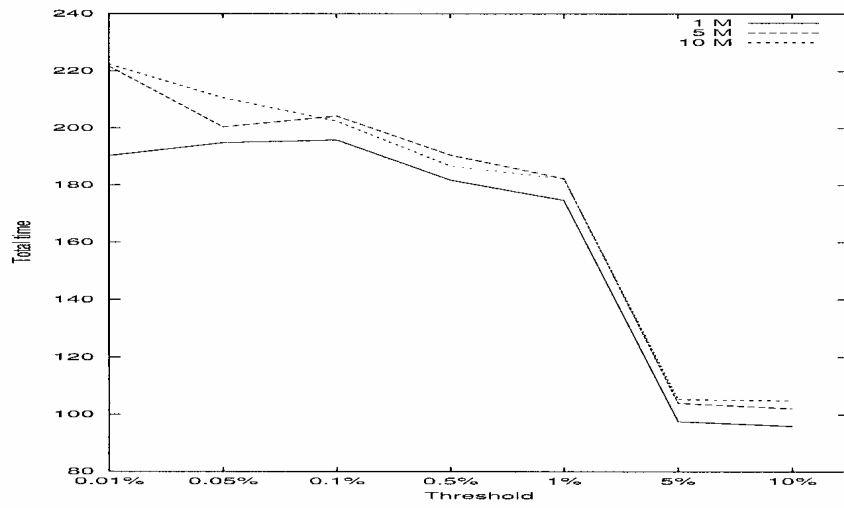


Fig. 7. Memory size vs. threshold: Total time in seconds.

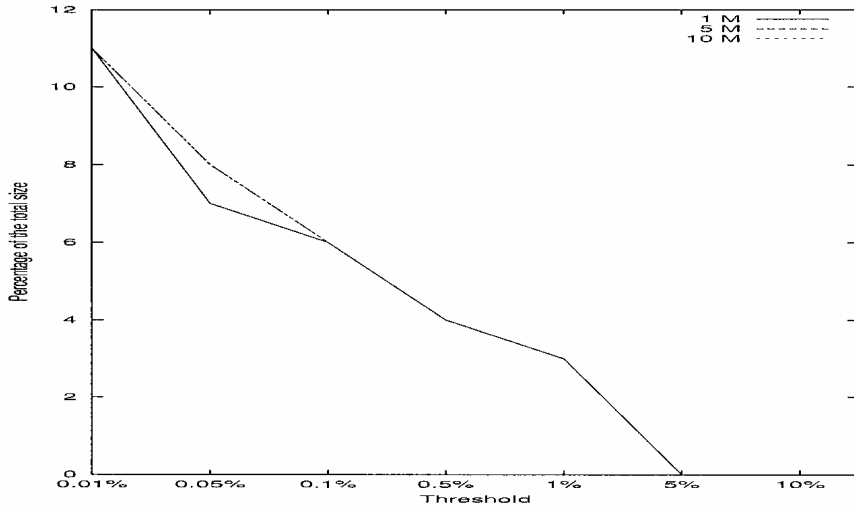


Fig. 8. Memory size vs. threshold: Percentage of data processed in the second phase.

Real dataset We ran the new algorithm on the 1999 KDD cup dataset (Att5 and Att6). Fig. 9 shows the total time for different threshold values. The behavior of the algorithm on real datasets comforts the general findings of the previous experiments. The total running time slowly decreases with the increase with the threshold values till a significant drop happen when the number of required I/O reaches to one scan.

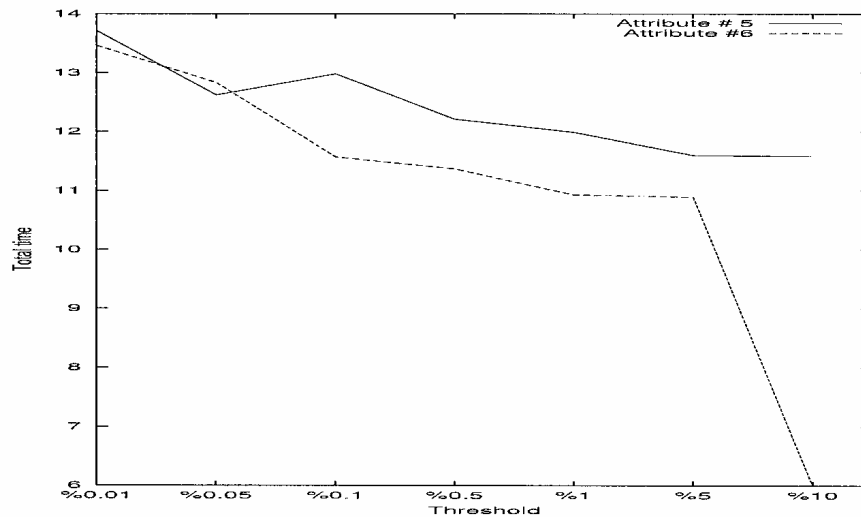


Fig. 9. Attributes number 5 and 6 of the real datasets: Total time in seconds.

We conclude from the analysis of Section 3 and the above experiments that the new algorithm is scalable with respect to the value of the threshold f ; it can efficiently handle relatively small values. For all the above experiments, we measured the I/O and computation times. The I/O generally consumes between 45% and 70% of the overall time. This fact offers an opportunity to further reduce the overall cost of the algorithm by overlapping the I/O with the computation.

Conclusion

We have presented an efficient algorithm for computing an important class of the iceberg queries which returns frequently occurring values from a set of attributes. It uses a set of estimates of quantiles of the input relation to focus the search of the query result. The new algorithm almost always requires two (or less) scans. It can compute the iceberg queries using one scan for some input. The new algorithm has nice properties that make it the algorithm of choice for solving the iceberg queries considered in this

work. It scales very well with respect to the relation size as well as the threshold. It is robust against the data distribution. Its memory and computational requirements are small.

As a future work, the I/O with the computation can be overlapped to further reduce the overall running time. Also, optimizing the memory accesses to better utilize the memory hierarchy may reduce the running time. One can use more quantiles to devise more aggressive pruning techniques. There is, however, a tradeoff between the pruning cost and the cost of Step 2 of Phase II which needs to be investigated further. It would be interesting to extend the new algorithm where the data elements have different weights.

References

- [1] The Third International Knowledge Discovery and Data Mining Tools Competition. Fifth ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 1999.
- [2] R. Agrawal, T. Imielinski, and A. Swami. Mining Associations Between Sets of Items in Massive Databases. Proceedings of the ACM SIGMOD Int'l Conference on Management of Data, pages 207-216, May 1993.
- [3] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. Proceedings of the 20th Int'l Conference on Very Large Databases (VLDB '94), September 1994.
- [4] R. Agrawal and A. Swami. A One-Pass Space-Efficient Algorithm for Finding Quantiles. Proceedings of the 7th Int'l Conference Management of Data (COMAD-95), December 1995.
- [5] K. AISabti, S. Ranka, and V. Singh. A One-Pass Algorithm for Accurately Estimating Quantiles for Disk-Resident Data. Proceedings of the Int'l Conference on Very Large Databases (VLDB '97), pages 346-355, August 1997.
- [6] S. D. Bay. The UCI KDD Archive [<http://kdd.ics.uci.edu>]. University of California, Department of Information and Computer Science, 1999.
- [7] K. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. In the Proceedings of 1999 ACM SIGMOD Int'l. Conference on Management of Data, pages 359-370, 1999.
- [8] M. Blum et al. Time Bounds for Selection. *Journal of Computers and Systems*, 7:4:448-461, 1972.
- [9] A. Broder, S. Classman, M. Manasse, and G. Zweig. Syntactic Clustering of the Web. In Proceedings of the 6th Int'l World Wide Web Conference, April 1997.
- [10] S. Chaudhuri and L. Gravano. Evaluating Top-A: Selection Queries. Proceedings of the 25th Int'l Conference on Very Large Databases (VLDB '99), pages 399-410, 1999.
- [11] D. Donjerkovic and R. Ramakrishnan. Probabilistic Optimization of Top n Queries. Proceedings of the 25th Int'l Conference on Very Large Databases (VLDB'99), pages 411-422, 1999.
- [12] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing Iceberg Queries Efficiently. Proceedings of the 24th Int'l Conference on Very Large Databases (VLDB '98), 1998.
- [13] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusam editors. *Advances in Knowledge Discovery and Data Mining*. The AAAI Press/The MIT Press, 1996.
- [14] R. W. Floyd and R. I. Rivest. Expected Time Bounds for Selection. *Communications of the ACM*, 18(3):165-172, 1975.
- [15] C. Fox. *Lexical Analysis and Stoplist in Information Retrieval Data Structures and Algorithms*. Prentice Hill, edited by W. Frakes and R. BaezaYates, 1992.

- [16] A. P. Gurajada and J. Srivastava. Equidepth Partitioning of a Data Set Based on Finding its Medians. Technical Report TR-90-24, Computer Science Dept., Univ. of Minnesota, 1990.
- [17] R. Jain and I. Chlamtac. The P2 Algorithm for Dynamic Calculation for Quantiles and Histograms Without Storing Observations. CACM, Vol. 28, No. 10:1076-1085, October 1985.
- [18] Y. Li and A. Jain. Classification of Text Documents. The Computer Journal, 41(8):537-546, 1998.
- [19] G. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate Medians and other Quantiles in One Pass and with Limited Memory. In the Proceedings of 1998 ACM SIGMOD Int'l. Conference on Management of Data, pages 426-435, 1998.
- [20] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Efficient Algorithms for Discovering Association Rules. KDD-94: AAAI Workshop on Knowledge Discovery in Databases, Seattle, Washington, pages 181-192, July 1994.
- [21] J. I. Munro and M. S. Paterson. Selection and Sorting with Limited Storage. Theoretical Computer Science, 12:315-323, 1980.
- [22] F. Olken. Random Sampling from Databases. Ph.D. Thesis, University of California at Berkeley, 1993.
- [23] R. Ramakrishnan and J. Gehrke. Database Management Systems. McGraw-Hill, 2nd edition, 2000.
- [24] G. Salton. The SMART Retrieval System- Experiments in Automatic Documents Processing. Prentice Hall, 1971.
- [25] G. Salton. A Theory of Indexing. Society for Industrial and Applied Mathematics, 1975.
- [26] S. Thomas, S. Bodagala, K. AISabti, and S. Ranka. An Efficient Algorithm for the Incremental Updation on Association Rules in Large Databases. In Proceedings of the 2nd Int'l Conference on Knowledge Discovery and Data Mining, 1997.
- [27] A. Grama V. Kumar, G. Karypis and A. Gupta. Introduction to Parallel Computing: Design and Analysis of Algorithms. The Benjamin/Cummings Publishing Company, Inc, 1994.
- [28] G.K. Zipf. Human Behavior and the Principle of Least Effort. Addison-Wesley, Reading, MA, 1949.

:

(/ / / /)

.