Novel Algorithms for Computing Medians and Other Quantiles of Disk-Resident Data

Lixin Fu and Sanguthevar Rajasekaran

Dept. of CISE, University of Florida, Gainesville

{lfu,raj}@cise.ufl.edu

Abstract. In data warehousing applications, numerous OLAP queries involve the processing of holistic operations such as computing the "top N", median, etc. Efficient implementations of these operations are hard to come by. Several algorithms have been proposed in the literature that estimate various quantiles of disk-resident data. Two such recent algorithms are based on sampling. In this paper we present two novel and efficient quantiling algorithms, Deterministic Bucketing (DB) and Randomized Bucketing (RB). We have analyzed the performance of DB and RB and extended the analysis of the sampling done in prior algorithms.

We have conducted extensive experiments to compare all these four algorithms. Our experimental data indicate that our new algorithms outperform prior algorithms not only in the overall run time but also in accuracy. The new algorithms can be used either as one-pass algorithms to accurately estimate quantiles or as algorithms for computing the quantiles exactly.

Key words: quantiles, OLAP queries, estimation of quantiles, very large datasets, selection, one-pass algorithms, sampling, randomized computing, holistic operations in databases, bucketing

1. Introduction

Given an input data set $S= \{k_1, k_2, \ldots ,k_N\}$ of size N and an integer i, the selection problem is to find the ith smallest element in S. The selection algorithms have many applications. In parallel and external sorting algorithms, one can use keys with some specific ranks to partition the input data set into sections of approximately the same size and then sort each section independently. In data

warehousing applications, some OLAP queries need several popular holistic operations [9] such as "*top N*", median, top 10% element, etc.

If the size of the input data set is very large, say $N \gg M$, where M is the size of internal memory, then we need to design external algorithms to solve the selection problem. In external algorithms, I/O operations, i.e., disk accesses take much more time than local computations. So they are also called I/O algorithms. This I/O bottleneck exists in other layers of the memory hierarchy as well. For a given input size and memory size, the key issue is to minimize the number of passes needed to solve the problem.

A simple average-case optimal algorithm for selection can be found in [5] that works as follows. Use one of the input keys as the partition element and partition the input into two. The first part consists of all the input keys less than the partition key and the second part consists of all the input keys greater than the partition key. After partitioning the input, identify the part that contains the key to be selected (i.e., the ith smallest element). Finally, perform an appropriate selection in this part. This algorithm runs in $\Omega(N^2)$ time in the worst-case. By using the median of medians as the partitioning element, we can modify this algorithm to a worst-case optimal algorithm [13]. These algorithms may need multiple passes when applied for external selection.

Many parallel selection algorithms have been devised for the CRCW PRAM model [5], [6] as well as for other models such as the mesh, the hyper-cube, etc. The reader is referred to [7] for a survey on this topic.

A frequently used paradigm for selection is the following [7]:

1) Get a sample T of size s from the input sequence S. Let $T = \{t_1, t_2, ..., t_s\}$ be the sample.

2) Sort T to get T' and compute two elements q and u in T', such that

   a) The target (i.e., the element to be selected) is included in $[q, u]$. We call q and u the lower bound and the upper bound, respectively.

   b) The number of elements in S that are in $[q, u]$ is small.

3) Sifting: compare each input key k_i in the input data set with q and u.

Define

$$T_1 = \{k_i \in S \mid k_i < q\}$$

$$T_2 = \{ k_i \in S \mid q <= k_i <= u\}$$

Let $n_1 = | T_1 |$, $n_2 = | T_2 |$

4) If the element to be selected is in T_2, perform an appropriate selection in T_2 using recursion or in-core algorithms.

T_2 ---> S

i- n_1 ---> i

Different algorithms have different flavors on how to sample and how to compute q and u.

Some of the works on selection are summarized now. Rajasekaran [2] gives a deterministic algorithm and a randomized algorithm on PDS (Parallel Disk Systems) which was introduced by Vitter [8]. Both algorithms use $O(N/DB)$ I/O read operations, where N is the input size, D is the number of parallel disks, and B is the block size. Clearly, these algorithms are asymptotically optimal. Furthermore, they have small constants. Munro and Paterson [4] have given a single pass algorithm for computing approximate quantiles. Alsabti, Ranka, and Singh [3] have designed a quantiling algorithm with error bounds. Manku, et al. [1] recently set up a uniform framework for the 1-pass quantiling algorithms and proposed a new algorithm with error guarantees. Manku, et al.'s algorithm dynamically exploits all the available buffers and needs less memory than the other two algorithms. Related works are also reported in [14], [15], and [16].

Given that several algorithms have been proposed in the literature for external selection, an important problem is to identify the most practical ideas. In this paper we analyze and experimentally compare four different algorithms. The first two algorithms in our list are the ones proposed by Rajasekaran [2] and Manku, et al. [1], respectively. It should be noted here that the original algorithm of Manku, et al., was proposed for computing approximate quantiles. We adapt this algorithm for exact selection by using the given rank error guarantee. The third and fourth algorithms are the ones we propose in this paper and are based on bucketing. The third algorithm uses deterministic bucketing. Though the worst case performance of this simple bucketing algorithm can be bad, its expected performance is good. Furthermore, we have designed a dynamic version of this bucketing algorithm that works well in the worst case. The fourth algorithm is based on randomized bucketing. Its worst case performance is good with high probability. It can also be expected to perform well in practice. It should be noted here that all these algorithms could either be used as one-pass algorithms for estimating quantiles or as exact algorithms for selection.

The rest of the paper is organized as follows. In section 2, we extend the analysis given in [2] for deterministic sampling. In section 3, the adaptation of Manku, et al.'s quantiling algorithm for

exact selection is given. We present our new bucketing algorithms in sections 4 and 5. Comparison and simulation results are summarized in section 6. Section 7 concludes the paper.

2. Deterministic Selection Algorithm of Rajasekaran

The idea behind the algorithm of Rajasekaran [2] is to use regular sampling (RS). RS used in [2] is an extension of the idea of Munro and Paterson. The idea is to obtain a layer of buffers of samples $b_0, b_1, \ldots, b_t$ from the input set S. Buffer b_0 is obtained directly from S. Bounds q and u are computed from the top buffer b_t. b_{i+1} is the k-regular sampling set from b_i. To obtain b_{i+1} from b_i sort b_i to get $b_i' = \{x_1, x_2, x_3, \ldots\}$. b_{i+1} consists of x_k , x_{2k}, …. Refer to k as the sample period.

For the external selection problem, the optimal case needs two passes: in the first pass, lower bound q and upper bound u are found. In the second pass, sift the input data to retain elements between the bounds. If the survivors fit into memory, then the target can be computed internally. Next, we will derive the relationships that must be satisfied to achieve optimality. An analysis for the case $N = (\sqrt{M})^c$, for any integer c, is given in [2]. We extend this analysis for $N=kM$ for any integer k.

Let j be the number of levels of sampling and let the sampling period k be the same at each level. First, we find the required relationships of the parameters to satisfy the following conditions:

(1) Bracketing: the target is contained between the bounds.

(2) The number of survivors is no more than M.

(3) The sample size at the top buffer is no more than M.

Theorem: To satisfy above conditions, the following must be satisfied:

(1) $d = k-1$. Here d is a sampling parameter. This parameter is used to pick the elements q and u.

(2) $k^j (3k-2)<(M+k)$

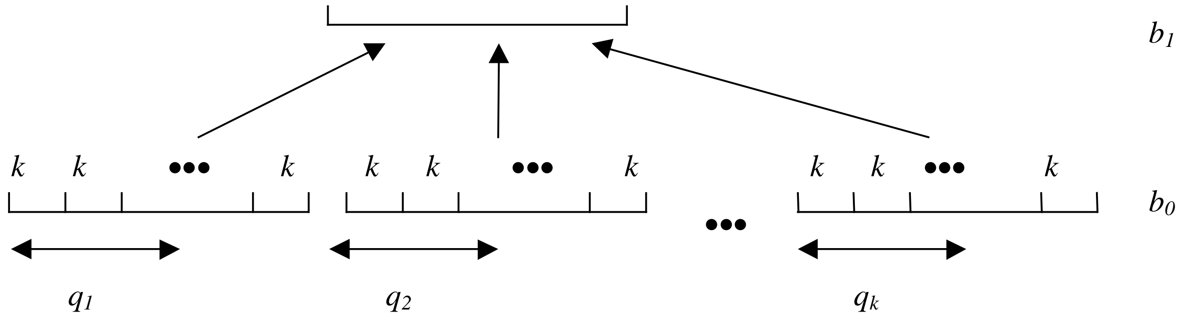
(3) $N<k^j M< M^2$

The following Lemma is needed in the proof.

Lemma: The element x with rank q in buffer b_1 will have rank in the range $((q-1)k, (q-1)k + k(k-1))$ in buffer b_0.

Since each sample in b_1 represents k elements in b_0, the element with rank q in buffer b_1 is larger than at least $(q-1)k$ elements in b_0. Suppose N elements are divided into k subsets $S_1, S_2, ..., S_k$ with the same size $N/k = M$ and there are $q_1, q_2, ..., q_k$ elements that are no larger than x respectively. Then, we have $q-1=\sum_{i=1}^{k} q_i$.

At each subset, there are (k-1) elements that are lager than x in the worst case for the next larger sample. So the rank is at most $(q-1)k+k(k-1)$.



Now return to the case with j levels. The idea is to pick two elements q and u from the top level (i.e., level j) buffer such that these elements bracket the element to be selected. Note that the expected rank of the element to be selected in buffer j is i/ k^j. If q and u are picked to have ranks around this expected value, then we could perhaps bracket the element we want. Let the elements q and u have ranks $i / k^j - d$ and $i / k^j + d$, respectively. Suppose the minimum and maximal positions in b_{j-1} of the two bounds are at positions $E, F, G,$ and H respectively. From the lemma, we have

$$E = (\frac{i}{k^j} - d - 1)k$$

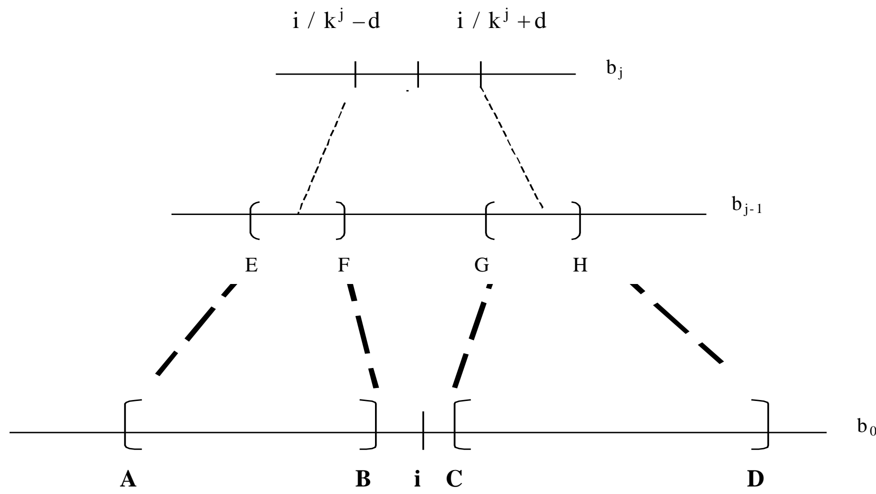
$$F = (\frac{i}{k^j} - d - 1)k + k(k-1)$$

$$G = (\frac{i}{k^j} + d - 1)k$$

$$E = (\frac{i}{k^j} + d - 1)k + k(k-1)$$

Similarly, we can proceed down level by level until we compute minimal and maximal rank positions A and B in b_0 of the lower bound $i / k^j - d$, minimal and maximal rank positions C and D in b_0 of the upper bound $i / k^j + d$ as follows.

$A = i-(d+1)\, k^j - k^{j-1} - \ldots - k^2 - k$
$B = i + k^{j+1} - (d+1)\, k^j - k^{j-1} - \ldots - k^2 - 2k$
$C = i+(d-1)\, k^j - k^{j-1} - \ldots - k^2 - k$
$D = i + k^{j+1} + (d-1)\, k^j - k^{j-1} - \ldots - k^2 - 2k$



$$B < i \rightarrow (d+1)k^j > k^{j+1} - k^{j-1} - \ldots - k^2 - 2k$$
$$\rightarrow d > k - 1 - \frac{1}{k} - \ldots$$
$$\rightarrow d = k - 1$$
$$i < C \rightarrow d > 1$$

$$D - A = k^{j+1} + 2dk^j - k = k^{j+1} + 2(k-1)k^j - k = k^j(3k-2) - k < M$$
$$\rightarrow k^j(3k-2) < (M+k)$$
$$N < k^j M < \frac{M(M+k)}{3k-2} < \frac{M(M+k)}{2k} < \frac{M}{2}\left(\frac{M}{k}+1\right) < M^2 \quad if\ k > 2$$

If these conditions are not satisfied, the algorithm can not give the exact answer internally after sifting.

Example: S = {1.007, 1.9, 1.013, 1.004, 1.006, 1.008; 1.5, 1.012, 1.005, 1.003, 1.002, 1.001; 1.7, 1.015, 1.010, 1.011, 1.009, 1.014}. N=18, M=6, k=3;

Top buffer is {1.003, 1.007, 1.011, 1.5, 1.7, 1.9}. i=6. d= k-1=2.

True target is 1.006. The estimated lower bound is negative infinity and the upper bound is 1.5. The number of survivors is 16 after sifting. Furthermore, the elements at the top buffer are far from the true quantiles.

From the theorem, if $N>M^2$, regular sampling algorithm will not be able to compute the true target internally after sifting. In fact, not only RS cannot, any other selection algorithm can not either in this case.

Theorem: If $N>M^2$, in the worst case, no algorithm can find two bounds and then use internal algorithm to find the exact answer from the survivors after sifting.

Proof: Suppose an algorithm finds (M-1) values as the boundaries. All the input data is divided into M subsets. If $N>M^2$, according to pigeonhole principle, there exists at least one subset whose size is larger than M. In the worst case, if the target is in that subset, the algorithm cannot solve the selection problem internally.

Suppose multiple selection queries are to be evaluated, we need to minimize the expected response time. Given input size $N \gg M$, the best we can do is to compute the approximate $2N/M$-quantiles and store them in array B of size M/2 in the first pass. In this way, the expected rank error is minimized. In the second pass, the exact histograms are computed and stored in another array C of size M/2 to give rank error guarantees.

3. Computing bounds by using approximate quantiles and error guarantees

Manku et. al. propose a generalized framework for selection which generalizes Munro-Paterson algorithm and the algorithm proposed in [3].

Manku, et al.'s algorithm is characterized by two parameters b(number of buffers), k (size of each buffer) and a set of operations: NEW, COLLAPSE, and OUTPUT. NEW fills the buffers from the input; COLLAPSE uses the data from a set of input buffers to compute one output buffer and empty the input buffers. OUTPUT is similar to COLLAPSE except that it outputs the final result.

Each Buffer is associated with a weight value and a level value. After the NEW operation, the buffer's weight and level values are assigned to 1. The level of the output buffer is one plus the level of the input buffers and the weight of output buffer is the summation of the input buffer weights. The input buffers of the COLLAPSE operation should be sorted in advance and have the same level value. This COLLAPSE process is similar to merging in the external merge-sorting algorithm. Each element in an input buffer is copied w times, where w is the weight of the buffer. All the elements in the input buffers together with their copies are sorted and arranged in rows of the same size, which is the weight of the output buffer. Each middle position element of every row is populated into the output buffer.

The process can be represented as a tree with nodes labeled by their weights. Each node represents a buffer. Manku, et al.'s algorithm differs from prior algorithms in the collapsing policy. When there is no empty buffer, a COLLAPSE operation is triggered. Otherwise, NEW operations are performed. This process can be regarded as iterative sampling (IS). The pseudo-code of the algorithm follows.

WHILE (not EOF) DO

    Fill *b* buffers initially and push them into a stack;
    From top down, find *c* buffers that have the same level value;
    COLLAPSE these *c* buffers into output buffer;
    Pop out *c* buffers from stack and empty them;
    Push the output buffer in stack

OUTPUT the final result from all the buffers in stack;

In the resulting buffer of OUTPUT operation, the approximate target position is at $\lceil q_1 kW \rceil$, where k is buffer size.

The paper of Manku et al also gives the upper bound of the rank difference between real target and output result. From the error guarantee, we can compute the bounds that bracket the target, thus adapting the approximate quantiling algorithm to the selection problem.

Max Rank Error $= (W-C-1)/2 + Wmax$, where

W: the summation of all COLLAPSE result buffers' weights

C: total number of COLLAPSE operations

Wmax: maximal weight of root's children

Define $q_1 = (i\text{-}Rank_Error)/n$, $q_2 = (i+Rank_Error)/n$, then

$$l = t[\lceil q_1 kW \rceil] \text{ and } u = t[\lceil q_2 kW \rceil]$$

Or $\pm\infty$ when the positions are out of range. These bounds are conservative based on the guarantees.

4. Deterministic Bucketing Algorithms (DB)

In this section, we propose two deterministic selection algorithms based on bucketing. The first algorithm is static and the second is dynamic.

4.1. Simple Bucketing (SB)

When the minimum and maximum values of the input data set are known and the data is not too skewed, the data can be divided evenly into ranges or buckets. For each bucket, an integer counter is used to count how many elements are in this bucket during the scanning of the input file. So an integer array is enough to keep the histograms of the buckets. To obtain the target bucket, sum up the histograms of the buckets until the summation is larger than or equal to the given value i. The bucket bounds will bracket the target and the rank error is not more than the histogram of the target bucket.

To get better estimations, the target bucket bounds play the roles of minimum and maximum and we redo the process with a changed value for i, which is i minus the summation of previous histograms. This needs an additional pass of input data but the coarse estimation can be returned in the first pass. The second pass is necessary only when the data is very skewed or an extremely good estimation is needed.

Furthermore, the time cost can be amortized. The goal is to set up an internal data structure through two passes of the input data set and then answer any number of queries in memory with high accuracy. The estimation is very close to the true target in terms of both the value and the rank.

First, an array C[0.. number of Buckets-1] is used to store the counts after scanning. Usually many buckets are empty; therefore some data compression is needed to free more memory for the second pass. The first compression technique is to use an array of (index, count) pairs (AP). When

there are many consecutive non-empty buckets, AR compression (Array of Ranges) is used. A range has three parts (start index, end index, count array). The count array records the counts in these buckets between two indexes. These two compression approaches are not always effective. To choose the right data structure requires a scan of the count array C.

Define

Z = number of zeros in C.

$$S = \sum (t-2)$$

The summation is over all consecutive non-empty bucket segments and t is the length of the segment.

If $Z > b/2$, use AP instead of array. b is the number of buckets.

If $S>0$, use AR instead of AP.

Example: Suppose b=40 and

C=[0 2 0 0 0; 0 3 1 2 71; 0 0 4 15 0; 0 25 0 0 0; 4 8 9 10 6; 0 0 0 0 0; 1 0 0 0 0; 0 0 0 0 0],

then Z=26, S=-1+2+0-1+3-1=2

Since $Z>b/2$, AP needs less memory than C. AP is:

$$\begin{pmatrix} 1 & 6 & 7 & 8 & 9 & 12 & 13 & 16 & 20 & 21 & 22 & 23 & 24 & 30 \\ 2 & 3 & 1 & 2 & 71 & 4 & 15 & 25 & 4 & 8 & 9 & 10 & 6 & 1 \end{pmatrix}$$

AR is:

$$\begin{pmatrix} [1,1] & [6,9] & [12,13] & [16,16] & [20,24] & [30,30] \\ C[1] & C[4] & C[2] & C[1] & C[5] & C[1] \end{pmatrix}$$

Each array in the second row of AR represents the count values in the corresponding interval at the first row. For example, C[4] = {3, 1, 2, 71}. It corresponds the interval [6, 9].

Since S>0, use AR instead of AP. The sizes of C, AP, and AR (in number of integers) are 40, 28, and 26 respectively.

If we can relax the "consecutive non-zero segments" condition to allow at most z consecutive zeros in a segment (z is usually 2 or 3), AR can be further compressed. For instance, if z=2, AR can be compressed into:

$$\begin{pmatrix} [1,1] & [6,16] & [20,24] & [30,30] \\ C[1] & C[11] & C[5] & C[1] \end{pmatrix}$$

The size is still 26 but has a small array (less number of nodes).

Once the saved space is freed, if necessary, one can further divide the dense buckets whose count values are larger than a threshold γ into sub-buckets. The histograms of the sub-buckets can be computed through the second pass of input data.

The freed space allocated to the dense buckets is proportional to their counts. Suppose the freed space is m. $B_1, B_2,\ldots, B_k$ are the dense buckets and $C_1, C_2,\ldots, C_k$ are their counts.

$$s = \sum_{i=1}^{k} C_i$$

Suppose $\delta_1, \delta_2,\ldots, \delta_k$ are the step lengths of these sub-buckets. Δ is the step-length of the first level division. Then

$$\delta_i = \Delta / (\frac{mC_i}{s}) = \frac{s\Delta}{mC_i}$$

Using these parameters, the count arrays (CA) $A_1, A_2,\ldots, A_k$ are set up for $B_1, B_2,\ldots, B_k$ by scanning the input data. From C (or AP, AR), CA, and target value i, find the target bucket and sub-bucket. The estimations of the target value and rank can also be computed.

This algorithm is easily modified to solve external sorting problem. From the histograms of the buckets, compute b bound values such that the segments have similar sizes. Cluster the input data and write back to disk the partitions. Then sort each partition individually. The sorting algorithm is easily parallelized because the loads are balanced.

In addition to solving the selection problem, the bucketing algorithm can also answer many other useful queries internally by exploiting C (or AP, AR) and CA. For example, summing up the histograms of the buckets and sub-buckets until the given element falls into the target bucket results in a good rank estimation of an element. The approximate number of elements in a given range can also be computed through the proper data structures proposed above.

Next, let us consider the worst case of the simple bucketing algorithm for out-of-core selection. Let MAX and MIN stand for the maximum and minimum values that any key can take. Divide the range [MIN-MAX] into M equal intervals (M being the core memory size). In one pass count the

number of keys in each interval and identify the interval containing the ith smallest element. Also count the number of elements falling in this interval. In the next pass, partition this interval into M parts and so on until the number of surviving keys is no more than M. When this happens, perform an appropriate selection using an internal selection algorithm.

In the worst case, this algorithm may need a large number of passes. We can design an input data set such that in **each** pass of the data, the bucketing algorithm can only rule out **two** elements. This input data set can be generated as follows:

To be simple, let $i = N/2$ (i.e. we want to find the medium). Suppose the data is represented as an array A[1..*N*].

Divide the data into M equal intervals $I_1, I_2, \ldots, I_M$, let A[1] = MIN, A[2]=MAX. Let all other elements be in one interval, say I_k. Then take these remaining elements (N-2 elements in all) as new input data and design the remaining elements of array A (A[3], A[4], and so on) recursively.

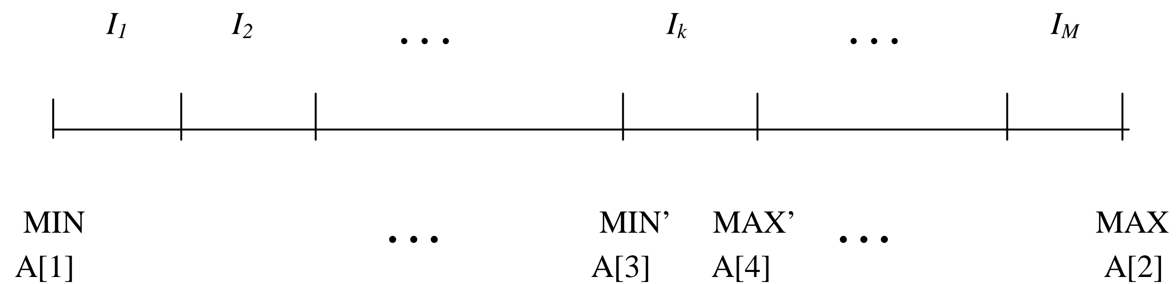
The input of the new reduced problem is:

$N \leftarrow N$-2;

MIN $\leftarrow$ minimum of the N elements;

MAX $\leftarrow$ maximum of the N elements;

We can take the target interval bounds as new MIN (MIN') and new MAX (MAX').



A[1] =0, A[2] = 1, A[3] =0.1, A[4] = 0.2, A[5] =0.11, A[6]=0.12,…,

A[49]=$0.1^{23}1$, A[50]= $0.1^{23}2$, A[51]=$0.1^{23}05$, where 1^k denotes k continuous one's. Of course, the array A can be arbitrarily permuted as the final input data.

Knowing how to design the worst case input data, one can compute the number of passes needed for the bucketing algorithm. Suppose p passes needed, i.e.

$N-2p<M, ==> p>(N-M)/2.$

This is certainly not a desirable result for the bucketing algorithm. But in most real-world applications such as if the data is distributed relatively evenly (for example, uniformly random data) or even if the data comes from a Gaussian distribution with not too large a variance, our experiments show that the bucketing algorithm has good performance. In addition, the bucketing algorithm is simple to implement and has value and rank error guarantees. So, given these error estimations, the users may decide whether to continue the next pass or not.

4.2. Dynamic Bucketing

To greatly reduce the number of passes required in simple bucketing in the worst case, we design a new algorithm called *Dynamic Bucketing*. Instead of compressing the data after passes, divide the whole process into h phases and applying some compression technique after each phase. In each phase, N/h elements are read and the histograms are computed. After this, apply some compression techniques given in the previous subsection and reclaim the space for later use. The available memory space is maintained and allocated to the dense buckets proportional to their histogram values. In the next phase, the histograms including those of the newly generated sub-buckets are computed. Repeat this process until the end of input file. In this way, the space allocation and data structure adapt dynamically to the input data distribution and input order.

5. Randomized Bucketing (RB)

Using random sampling techniques to choose the approximate quantiles as bucket boundaries is a simpler and effective way to solve the selection problem.

In *Randomized Bucketing* (RB), choose M elements randomly from the input data set in the first pass. One could use the sorted sample keys to define bucket boundaries. It is easy to show that the size of each bucket is no more than $O(N/M \log N)$ with high probability [11][12]. We can obtain even better partitions as follows. Choose M random elements as before and sort them. After sorting these elements, choose every other element as a sample element. We take these $M/2$ elements as bucket boundaries. In the second pass, compute and store the histograms of the buckets, i.e., *count* values into an array C of size $M/2$. Sum up the counts of C until the summation is larger than i. We can get the lower bound and upper bound of the target with rank error guarantee.

In fact, the above technique can be generalized to sample every k^{th} element of the sorted sample and use them to define bucket boundaries. In this case we can show that with high probability the size of each bucket is *O(Nk/M)*. We now give a proof for this fact.

Theorem: Let *X* be an input sequence of size *N*. Let $S = q_1, q_2, ..., q_s$ be a sorted sample from *X*. Say we pick every k^{th} element of this sequence to form the sequence $R = r_1, r_2, ..., r_{s/k}$. The sequence *R* partitions *X* into (*s/k+1*) parts. The size of each part is *O(Nk/s)* with high probability.

Proof: Consider any ordered subsequence of *X* of length *y*. We want to compute the probability that all these elements will belong to a single part of *X*. This will happen if all these elements have a value in the range $[r_i, r_{i+1}]$ for some I .This implies that from out of the *y* elements under consideration, exactly *k* elements belong to *S*. The rest of the elements of *S* have been picked from out of *N*-*y* elements. The probability of this happening is $\frac{\binom{N-y}{s-k}\binom{y}{k}}{\binom{N}{s}}$. This probability can be shown to be no more than $N^{-\alpha}$, for any fixed $\alpha \geq 1$ for *y=O(Nk/s)*. Choices for *k* and *s* are *O(1)* and N^{ε}, for any fixed $\varepsilon < 1$, respectively.

We have two versions of the randomized bucketing algorithm, viz., RB1 and RB2. RB1 makes only one pass through the data and picks a suitable sample. From the sample it identifies approximations to the quantiles that we are interested in. RB2 makes one pass through the data to pick a sample and identify elements that bracket the target. In the second pass it computes the selection element exactly. In our experiments RB1 was the fastest from among all the algorithms tested. Its accuracy was not great but acceptable. RB2 took more time than RB1 but produced more accurate estimates. RB1 is an excellent choice when one is interested in a quick estimate of quantiles. Note that the performance of RB1 is independent of the input distribution and permutation.

Similar sampling techniques have been explored before. See for example [7], [10],[12].

6. Simulation Results

We have done extensive experiments to compare the performance of the algorithms discussed thus far in the paper. These experiments were done on IBM Aptiva E Series 545 PC with 96M memory and 450MHZ Pentium III processor. The operating system is Windows 98. We used the file system to access the files on disk. All the algorithms are implemented in Java language.

6.1. Data Sets and Overview of the Simulations

We used synthetic data sets in the simulations. After generating the data sets, we wrote them as text files on the disk. These data files were the inputs to the algorithms. Using uniform and normal random generators, we generated the following three data sets.

Data set D1 contains N permuted integers from 0 to N-1. This set represents the rank space. The consecutive integers are broken into large blocks. The elements are randomly permuted within blocks. The permuted blocks are then written to disk as input data. Block size and permuted times are two parameters.

Data set D2 contains uniformly distributed random data in [0, 1]. The size is N. Random real numbers are written to disk one by one.

Data set D3 is a normally distributed random data set with variance 1 and mean 0. The size is N.

We have implemented and experimented with all the four algorithms: deterministic selection algorithm using regular sampling (RS) [2], modified approximate quantiling algorithm i.e. iterative sampling (IS) [3], and our selection algorithms using buckets: deterministic bucketing (DB) and random bucketing (RB). We compared the computing times and accuracy of these algorithms under varying input data sizes, selection queries, and memory sizes to explore their behavior and performance. These algorithms were tested as one-pass algorithms for estimating quantiles as well as algorithms for exact selection. “Accuracy” refers to the number of survivors after “killing” the elements that lie beyond the boundaries provided by the algorithms (when used as one-pass algorithms).

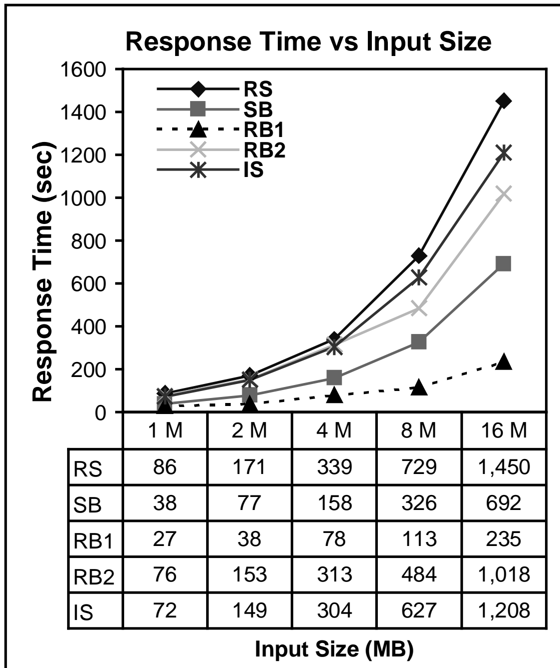
6.2 Varying Input Data Size

In this subsection, we explored the response times and accuracy of the algorithms with the increase of input data size namely the number of keys. We fixed the element to be selected (i.e., median) and the memory size (200KB) and increased the input size from 1 MB to 16 MB in an exponential fashion. Data sets D0, D1 and D2 are used to reflect different input data distributions. In D0, the parameters block-size and permuted-times are chosen to be 2048 and 20, respectively.

We implemented a two-level RS algorithm and used two buffers, each of size $M/2$. As in Section 2, let sampling period k be $2N/M$ and d be $(k\text{-}1)$. In IS, we set b=7. This algorithm gives very conservative rank error bounds. In our experiments, setting d to be d_{RS}/b is enough to derive bounds that can bracket the target.

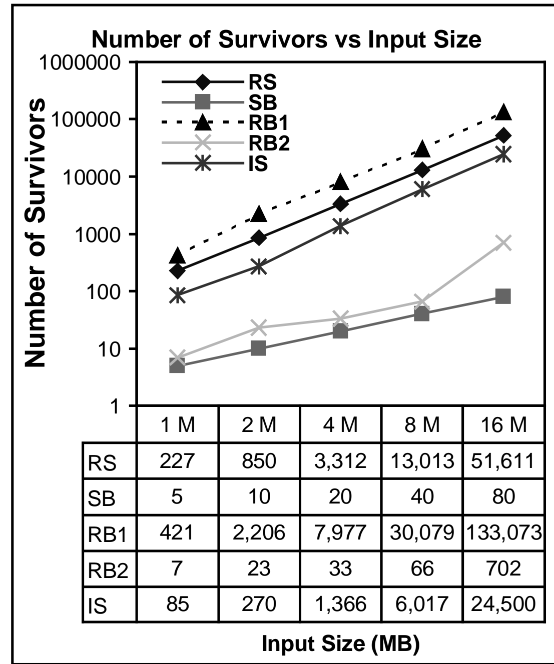
For DB, we implemented the simple bucketing algorithm (SB) assuming bounds MIN and MAX are available. We assign [0,N], [0,1], [-10,10] to the boundaries [MIN, MAX] in data set D0, D1, D2, respectively. In case of unknown MIN and MAX, dynamic bucketing and appropriate compression techniques may be used. In randomized bucketing (RB), we first scan the input data set and include each element in the sample with probability (M/N). Once the sample set is formed in the first pass of input data, we can estimate the quantiles and lower and upper bounds. Our experiments show that when we set $d= 5d_{RS}$, the bounds can bracket the target. We denote the first pass estimation algorithm as RB1. RB2 is the version of the algorithm that samples and figures out the histograms of the various buckets (and is a two-pass algorithm).

Figure 1 to Figure 6 summarize the results. For all the data sets, SB and RB have clear improvement on the response times over RS and IS. The accuracy of our bucketing algorithms (SB and RB2) is much better (by 1-2 orders of magnitude) than previous algorithms. Generally, IS is better than RS in terms of response time and accuracy. It was noted that the accuracy of SB is better than RB2 for uniform random data but the accuracy of RB2 is better for normal random data. In randomized bucketing, the data is partitioned into intervals in rank space, i.e., there are approximately the same number of elements in these intervals. In SB, the intervals have equal length but the intervals need not have the same (or nearly the same) number of elements in them. The 1-pass randomized bucketing algorithm (RB1) is by far faster than all the other algorithms though its accuracy is not good. When a fast and coarse estimation is called for, RB1 is an excellent choice.



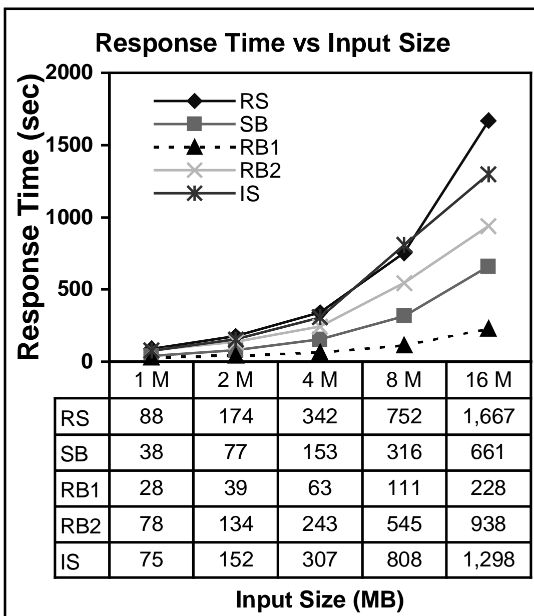
	1 M	2 M	4 M	8 M	16 M
RS	86	171	339	729	1,450
SB	38	77	158	326	692
RB1	27	38	78	113	235
RB2	76	153	313	484	1,018
IS	72	149	304	627	1,208

Figure 1: Response Times for DS0.



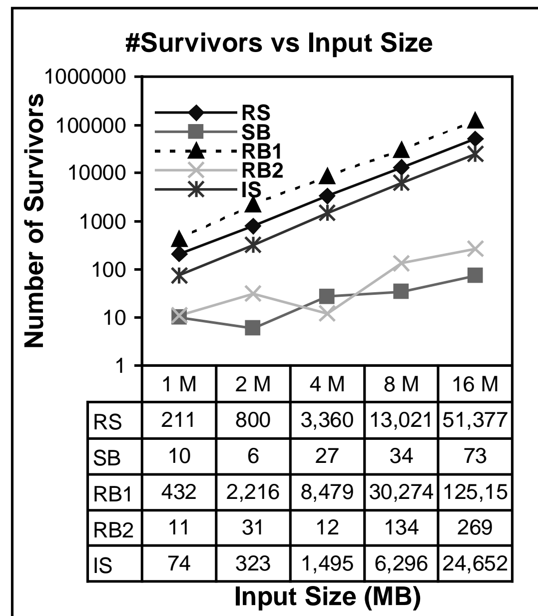
	1 M	2 M	4 M	8 M	16 M
RS	227	850	3,312	13,013	51,611
SB	5	10	20	40	80
RB1	421	2,206	7,977	30,079	133,073
RB2	7	23	33	66	702
IS	85	270	1,366	6,017	24,500

Figure 2: Number of Survivors for DS0.



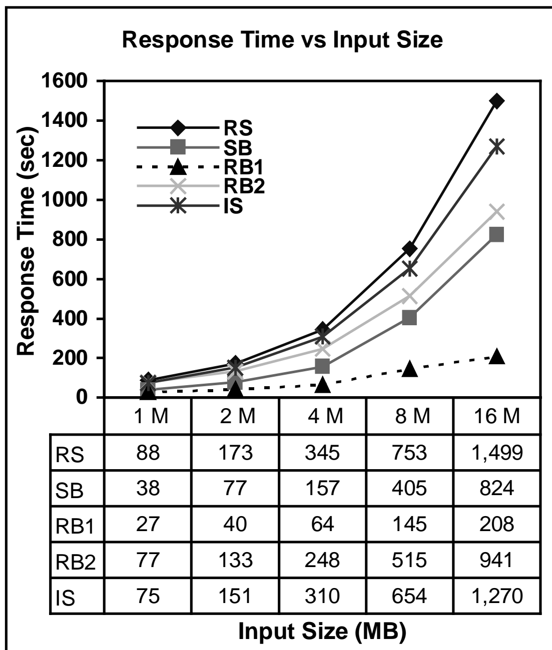
	1 M	2 M	4 M	8 M	16 M
RS	88	174	342	752	1,667
SB	38	77	153	316	661
RB1	28	39	63	111	228
RB2	78	134	243	545	938
IS	75	152	307	808	1,298

Figure 3: Response Times for DS1



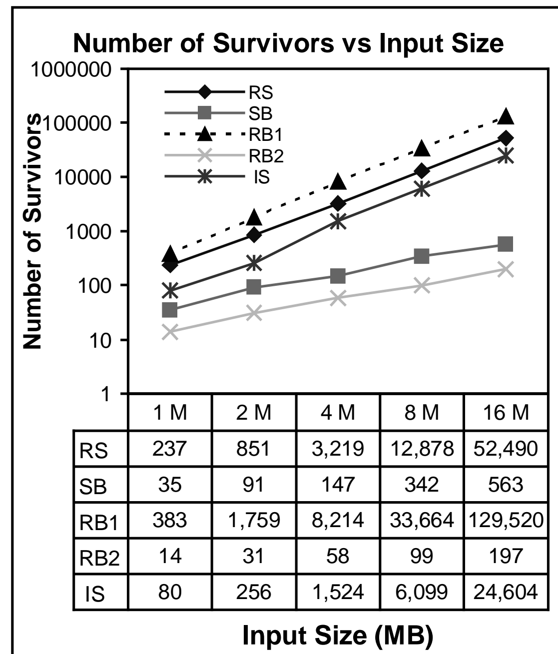
	1 M	2 M	4 M	8 M	16 M
RS	211	800	3,360	13,021	51,377
SB	10	6	27	34	73
RB1	432	2,216	8,479	30,274	125,15
RB2	11	31	12	134	269
IS	74	323	1,495	6,296	24,652

Figure 4: Number of Survivors for DS1.



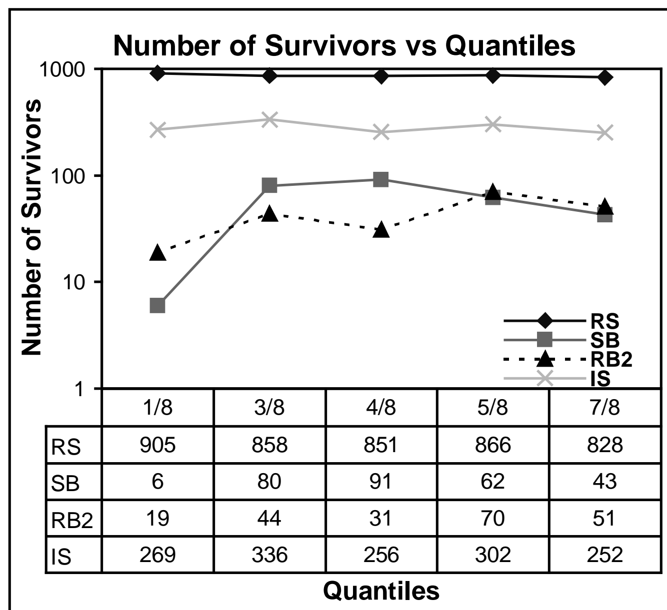
	1 M	2 M	4 M	8 M	16 M
RS	88	173	345	753	1,499
SB	38	77	157	405	824
RB1	27	40	64	145	208
RB2	77	133	248	515	941
IS	75	151	310	654	1,270

Figure 5: Response Times for DS2.



	1 M	2 M	4 M	8 M	16 M
RS	237	851	3,219	12,878	52,490
SB	35	91	147	342	563
RB1	383	1,759	8,214	33,664	129,520
RB2	14	31	58	99	197
IS	80	256	1,524	6,099	24,604

Figure 6: Number of Survivors for DS2.



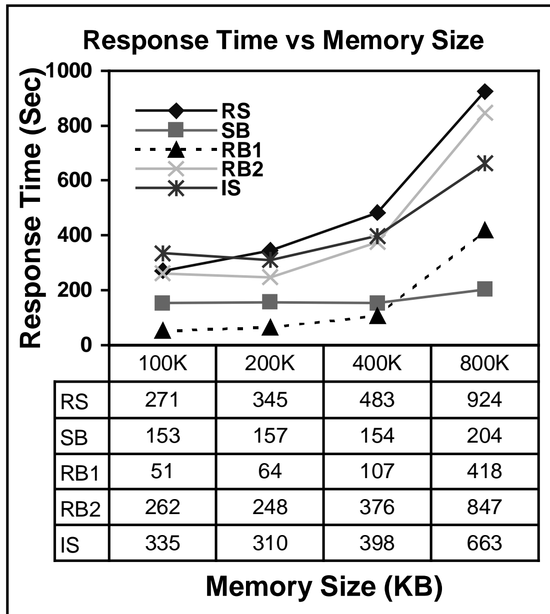
	1/8	3/8	4/8	5/8	7/8
RS	905	858	851	866	828
SB	6	80	91	62	43
RB2	19	44	31	70	51
IS	269	336	256	302	252

Figure 7: Accuracy for Normal Random Data.

6.3 Varying Selection Queries and Memory Sizes

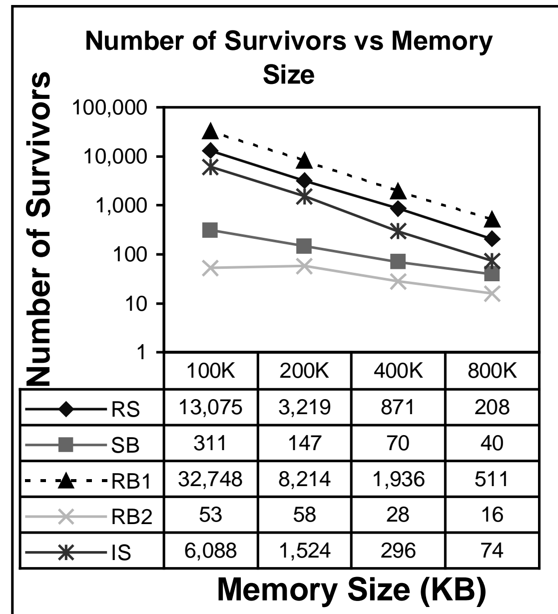
In this subsection, we fixed input sizes and changed the quantiling positions and memory sizes. In the first set of simulations, the input size was 2MB and the memory size was 20KB but the query positions changed from $N/8$ to $7N/8$. We observed that the response times were virtually the same. Since the times are predominantly determined by the time to set up the data structures (top buffer in RS, final output buffer in IS, sample set in RB1, and the statistics of the intervals in SB and RB2), different query positions do not influence the response times much. So we do not include the response times graph in the paper. As for the accuracy, we did experiments using the data set D2. From Figure 7, we can see the degrading of the performance in SB for queries close to the median.

Figures 8 and 9 show the response times and the accuracy of the algorithms with respect to change in memory size. Input size (4MB) and the element to be selected (median) were fixed while the memory size was increased from 100KB to 800KB.



	100K	200K	400K	800K
RS	271	345	483	924
SB	153	157	154	204
RB1	51	64	107	418
RB2	262	248	376	847
IS	335	310	398	663

Figure 8: Response Times for DS2.



	100K	200K	400K	800K
RS	13,075	3,219	871	208
SB	311	147	70	40
RB1	32,748	8,214	1,936	511
RB2	53	58	28	16
IS	6,088	1,524	296	74

Figure 9: Accuracy forDS2.

The response times increase with memory sizes but the accuracy increases quickly. In these cases, I/O times are the same. For RS and IS, the main task of CPU is sorting. Sorting in larger blocks spends more time. For example, sorting N keys takes $O(N \log N)$ time while sorting by k segments takes $O(kn/k \log(n/k)) = O(n(\log n - \log k))$. In bucketing algorithms, computing more

intervals takes more time. The accuracy of RS and IS approach those of bucketing algorithms when the memory size increases.

In summary, when MIN and MAX of input data are known and the data is approximately evenly distributed, SB is the best choice. When the data is very skewed and sparse, randomized bucketing is the right solution for quantiles.

7. Conclusions

This paper has extended the analysis of two existing algorithms namely regular sampling (RS) and iterative sampling (IS). We have designed and analyzed two new quantiling and selection algorithms namely deterministic bucketing (DB) and randomized bucketing (RB). We have developed two versions of each of these bucketing techniques. All these algorithms have been experimentally compared over a variety of input distributions, input sizes, memory sizes, etc. Our extensive simulations on various data sets clearly show that our new algorithms outperform prior algorithms in terms of response times and quantile estimating accuracy.

References

[1] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, “Approaxiate Medians and other Quantiles in One Pass and with Limited Memory”, *Proc. ACM SIGMOD Conference, 1998.*

[2] S. Rajasekaran, “Selection Algorithms for Parallel Disk Systems”, *Proc. International Conference on High Performance Computing, 1998*. Also to appear in *Journal of Parallel and Distributed Computing.*

[3] K. Alsabti, S. Ranka, and V. Singh, “A One-Pass Algorithm Accurately Estimating Quantiles for Disk-Resident Data”, *Proc. 23rd VLDB Conference*, Athens, Greece, 1997.

[4] J. I. Munro and M. S. Paterson, “Selection and Sorting with Limited Storage”, *Theoretical Computer Science*, vol. 12, pp. 315-323, 1980.

[5] E. Horowiz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*, W. H. Freeman Press, 1998.

[6] J. J'a J'a, *Parallel Algorithms: Design and Analysis*, Addison-Wesley Publishers, 1992.

[7] S. Rajasekaran, “Sorting and Selection on Interconnection Networks”, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 21,1995, pp. 275-296.

[8] J. S. Vitter and E. A. M. Shriver, “Algorithms for Parallel Memory I: Two-Level Memories”, *Algorithmica*, 1994.

[9] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals,” *Data Mining and Knowledge Discovery*, vol. 1, pp. 29-53, 1997.

[10] J.H. Reif and L.G. Valiant, “A Logarithmic Time Sort for Linear Size Networks,” *Journal of the ACM, 34(1),* 1987, pp. 60-76.

[11] S. Rajasekaran and J.H. Reif, “Derivation of Randomized Algorithms for Sorting and Selection,” in *Parallel Algorithms Derivation and Program Transformation,* Kluwer Academic Publishers, 1993, pp. 187-205.

[12] R. Motwani and P. Raghavan, *Randomized Algorithms,* Cambridge University Press, 1995.

[13] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, R.E. Tarjan, “Time Bounds for Selection,” *JCSS 7,* 1973, pp. 448-461.

[14] G. Piatetsky-Shapiro, “Accurate Estimation of the Number of Tuples Satisfying a Condition,” *Proc. ACM SIGMOD Conference,* 1984.

[15] V. Poosala and Y.E. Ioannidis, P.J. Haas, and E.J. Shekita, “Improved Histograms for Selectivity Estimation of Range Predicates,” *Proc. ACM SIGMOD Conference,* 1996, pp. 294-305.

[16] R. Jain and I. Chlamtac, “The P^2 Algorithm for Dynamic Calculation for Quantiles and Histograms without Storing Observations,” *CACM 28,* 1985, pp. 1076-1085.