Evaluating holistic aggregators efficiently for very large datasets

By: Lixin Fu, and Sanguthevar Rajasekaran

Lixin Fu, Sanguthevar Rajasekaran**, Evaluating Holistic Aggregators Efficiently for Very Large Datasets,** The VLDB Journal, Volume 13, Number 2 (May 2004), pp. 148-161.

Abstract:

In data warehousing applications, numerous OLAP queries involve the processing of holistic aggregators such as computing the “top n,” median, quantiles, etc. In this paper, we present a novel approach called dynamic bucketing to efficiently evaluate these aggregators. We partition data into equiwidth buckets and further partition dense buckets into sub-buckets as needed by allocating and reclaiming memory space. The bucketing process dynamically adapts to the input order and distribution of input datasets. The histograms of the buckets and subbuckets are stored in our new data structure called structure trees. A recent selection algorithm based on regular sampling is generalized and its analysis extended. We have also compared our new algorithms with this generalized algorithm and several other recent algorithms. Experimental results show that our new algorithms significantly outperform prior ones not only in the runtime but also in accuracy.

Keywords: Quantiles –Dynamic bucketing –Aggregation

Article:

1 Introduction

In decision support systems, due to large sizes of data warehouses, users frequently submit queries involving aggregation rather than specific details of the records. Gray et al. [5] categorize the aggregators into three classes: distributive aggregators such as SUM, COUNT, MIN/MAX, algebraic aggregators such as AVG, and holistic aggregators such as median, “top n”, selection, percentiles/quantiles, etc. Given input dataset S that contains N elements $k_1, k_2, \dots, k_N$ and an integer i, the *selection problem* is to find the element with rank i in S (i.e., the i-th smallest element). To answer a “top n” query, one should compute the n largest elements in S. The p-quantiles of S are the elements of ranks jN/p, $j = 1, 2,\dots, p — 1$ in S. The computation of the median is a special case of selection problem where i is equal to $N/2$. In this paper, we focus on selection problems for large disk-resident datasets.

The holistic aggregators are widely used to obtain summary information in very large data warehouses among many other important applications. In parallel and external sorting algorithms, for example, one can use keys with specific ranks to partition the input dataset into sections of approximately the same size and then sort each section independently. In query optimizations of databases, the approximate quantiles are useful in estimating the selectivity of queries.

The internal selection problem has been well studied. A simple average-case optimal algorithm for selection described in [9] works as follows. Use one of the input keys as the partition element and partition the input into two parts. The first part consists of all the input keys less than the partition key, and the second part consists of all the input keys larger than the partition key. After partitioning the input, identify the part that contains the key to be selected (i.e., the i-th smallest element). Finally, perform an appropriate selection in that part recursively. This algorithm runs in $\Omega(N^2)$ time in the worst case. By using the median of medians in 5-element groups as the partitioning element, we can modify this algorithm to a worst-case optimal algorithm [3]. These algorithms may need multiple passes when applied to disk-resident data. Many parallel selection algorithms have been devised for the CRCW PRAM model [9,11] as well as for other models such as the mesh, the hypercube, etc. The reader is referred to [17] for a survey on this topic. In addition, more examples of randomized selection algorithms can be found in [13,19,20].

If the size of the input dataset is very large, say, $N \gg M$, where M is the size of available internal memory in terms of number of elements stored, then we need to design external memory algorithms to solve the selection problem. In external memory 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 main goal is to minimize the number of passes needed to solve the problem. In data warehousing and OLAP contexts, many algorithms (e.g., [7,8,22]) are for distributive and algebraic aggregators. Efficient external memory algorithms for holistic aggregators are hard to come by. Some external memory algorithms use sampling techniques. A frequently used paradigm for selection is the following [17]:

1. Obtain a sample T of size s from the input sequence S using some sampling techniques. Let $T = \{t_1, t_2,...,t_s\}$ be the sample.

2. Sort T to T' and compute two elements l and u in T' such that

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

(b) the number of elements in S that are also in $[l, u]$ is small. These elements are called *survivors*.

3. Sifting: compare each input key *ki* in the input dataset with l and u.

   Define

   $T1 = \{k_i \in S | k_i < l\}$
   $T2 = \{k_i \in S | l \leq k_i \leq u\}$.

   Let $n_1 = |T1|$, $n_2 = |T_2|$.

4. Perform an appropriate selection in T_2 using recursion or in-core algorithms, where S and $i - n_1$ are replaced with T_2 and i, respectively.

Different algorithms have different flavors on how to sample specifically and how to compute l and u. Rajasekaran [18] gives a deterministic algorithm and a randomized algorithm on a *parallel disk model* (PDM). The PDM was introduced by Vitter [21]. Munro and Paterson [14] have given a single-pass algorithm for computing approximate quantiles. Alsabti et al. [2] have designed a quantiling algorithm with error bounds. Manku et al. [12] recently set up a uniform framework for the single-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 algorithms in [2,14]. Related work is also reported in [15,16].

Feder et al. proposed a very interesting problem: selection (e.g., median computation) with uncertainty [4]. The input N elements K_j can fluctuate in intervals $K_j \in I_j$, respectively (hence "with uncertainty"), and query of the their true values k_j has an associated cost C_j, where $j = 1, 2, .. ., N$. The problem is how to compute the i-th smallest element with precision δ such that total costs of the queries are minimal. They obtained a polynomial-time algorithm on the offline using linear programming. However, this algorithm does not have I/O-cost analysis and is infeasible when N is extremely large.

Online aggregation algorithms have the advantage that users can control the flow of execution on the fly and get good estimations of the aggregates. Agrawal and Swami [1] describe a single-pass online algorithm for finding quantiles. This incremental algorithm is insensitive to a priori data distribution and data input order, but there exist serious pathological examples in which the errors are large and not acceptable. Three additional heuristics are given but do not necessarily improve accuracy. A well-known online algorithm called P^2 [10] also maintains a list of "markers" as estimations of quantiles. They are dynamically updated as the new observations are processed. Algorithm P^2 does not have error guarantees. Hellerstein et al. [6] proposes another online aggregation method in which estimations are improved when more observations are available.

In this paper, we propose new online algorithms based on bucketing. Since the worst-case performance of a simple equiwidth bucketing algorithm can be poor because input data distribution can be very skewed (i.e., some buckets have extraordinarily high frequencies), we have designed a dynamic bucketing algorithm that works well in the worst case. We partition data into equiwidth buckets and further partition dense buckets into subbuckets as needed by reclaiming and allocating memory space. The bucketing process dynamically adapts to the input order and distribution of input datasets. The histograms of the buckets and subbuckets are stored in our new data structure called the *structure trees*. After the structure tree is initialized, to solve the selection problem we can traverse its buckets and subbuckets and sum up the histograms until the target is reached. Notice that in addition to efficiently evaluating holistic aggregators, this algorithm can be easily modified into efficient new algorithms for other important problems such as answering range queries, clustering large datasets, etc. We also give a new *randomized bucketing* algorithm whose worst-case performance is good with high probability.

We generalize the algorithm proposed by Rajasekaran [18] to allow a general sampling period. Through the analysis of the generalized algorithm, we obtain an interesting and important result – deriving a set of conditions under which the optimality of finding an *exact* target in two passes is achieved. We also adapt the approximate quantiling algorithm given by Manku et al. [12] to exact selection by using the given rank error guarantee. The generalized and modified algorithms are called *regular sampling* and *iterative sampling*, respectively, and their description and analysis are given in Sect. 5.

The rest of the paper is organized as follows. In Sect. 2, we describe the traditional equiwidth bucketing approach, its drawbacks, and our motivation for developing a dynamic bucketing approach. We present our new bucketing algorithms in Sects. 3 and 4. Comparison and simulation results are summarized in Sect. 6. Section 7 concludes the paper.

2 Simple bucketing and data compression

Though many algorithms are comparison-based, the value-based approach, in which elements are put into bins or "buckets" according to their values, has been somewhat neglected. This bucketing approach has many advantages if implemented appropriately. For example, we may only need to store the histograms of the buckets without saving their actual values to obtain aggregate information from which we can learn interesting patterns of data distribution. In this section, we will analyze such a bucketing algorithm and explain the motivation for exploring more sophisticated and effective new dynamic algorithms.

2. 1 Simple bucketing

If the minimum and maximum of the input dataset are known and the data are not too skewed, the elements can be directed into evenly divided ranges or buckets. For each bucket, an integer counter is used to count how many elements are in the bucket during the scanning of the input file. So an integer array is enough to keep the count values of the buckets (also called *histograms*). 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 will not be more than the count of the target bucket. The average of the two bounds serves as an approximation of the true target. We term this algorithm *simple bucketing* (SB), which is also widely known as the equiwidth histogram scheme.

10	2	80	82	1.105	88	9	1.104	82.6	83.4
1.19	86	1.11	2.5	85	1.18	1.17	1.12	7	1.107
1.1	31	1.109	1.1055	1.13	82.8	82.72	82.9	1.102	1.108
82.7	82.4	95	1.15	1.16	1.5	1.4	2.7	100	0
5	4	6	8	1.106	1.1066	82.5	1.14	6.3	1.1077

Fig. 1. Input data

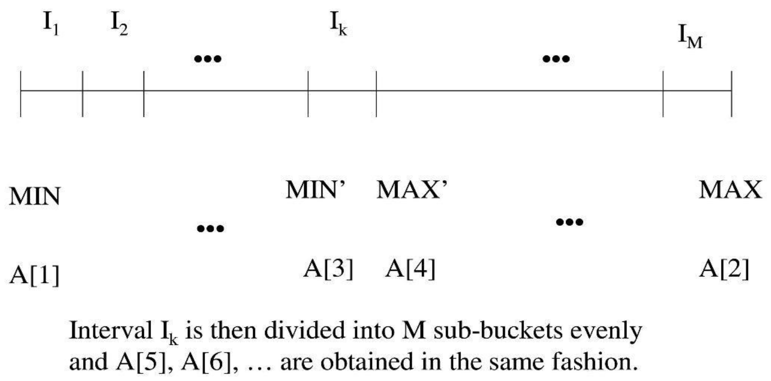

I_1
I_2
I_k
I_M
MIN
MIN'
MAX'
MAX
A[1]
A[3]
A[4]
A[2]
Interval I_k is then divided into M sub-buckets evenly
and A[5], A[6], … are obtained in the same fashion.

Fig. 2. Simple bucketing may need a large number of passes

We will use a running example to illustrate the algorithms discussed in this paper. Suppose the input dataset shown in Fig. 1 has 50 elements (N = 50) in [0, 100] and the median element 2.5 (or 2.7) is to be found or estimated.

If we use SB with a *count array* (CA) of size 20 for the running example, the histograms are ⟨27,6,1,0,0,0,1,0,0,0,0,0,0,0,0,0,10,3,0,2⟩. Since the first bucket [0, 5) contains 27 elements, it is the target bucket and the estimated median is 2.5.

To get a better estimation or exact target, the target bucket bounds play the roles of minimum and maximum and we redo the process with an updated 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 are very skewed.

However, in the worst case the SB algorithm for out-of-core selection may need multiple passes when input data are indeed highly skewed. We will design a somewhat artificial extreme case where SB needs a large number of passes. Let MAX and MIN denote the maximal and minimal values of the input data. Divide the range [MIN, MAX] into M equal intervals. In one pass, count the number of keys in each interval and identify the interval containing the i-th smallest element. Also, count the number of elements falling in this interval. In the next pass, partition this interval into M parts and get histograms. Repeat this process until the number of surviving keys is no more than M. When this happens, perform an appropriate selection using an internal selection algorithm.

We can design an input dataset such that in *each* pass of the data, the SB algorithm can rule out only *two* elements. This input dataset can be generated as follows. For simplicity's sake, let $i = N/2$ (i.e., we want to find the median). Suppose the input data are represented as an array A[1 ... N]. Figure 2 shows the process of designing a worst-case scenario. Divide the data into M equal intervals I_1, I_2, .. ., I_M and 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 $\leftarrow$ the new minimum of the surviving elements
MAX $\leftarrow$ the new maximum of the surviving elements.

Let us look at a very simple example, where $M = 10$, $N = 51$, $i = 26$. The input data are generated as follows. 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] = $0.1^{24}5$, where 1^k denotes k continuous 1s. Of course, the array A can be arbitrarily permuted as the final input data. A[5 1] is in between A[49] and A[50] and is the target (median) of A[1 ... 51]. Knowing how to design the worst-case input data makes it possible to compute the number of passes needed for the simple bucketing algorithm. Suppose h passes are needed, i.e., $N - 2h < M, \Rightarrow h > (N - M)/2$.

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

2.2 Motivation

One problem of the SB algorithm is that the minimal and maximal values may be unknown in advance. Furthermore, data may be very skewed when clusters with large histograms called *dense buckets* occur. The previous subsection constructed a pathological example in which a large number of passes were required to find the medium of a large dataset. Another drawback of SB is that some values are extremely large or small (e.g., - 1 billion and 1 billion) and are known as “wild observations.” If two of them serve as boundaries, the vast majority of buckets will be empty.

To fix the “data skewedness” problem, we process the in-put data in phases. After each phase, the dense buckets are further divided into finer subbuckets and their histograms up-dated by examining more observations. If necessary, the sub-buckets can be further divided, forming a granular hierarchy of buckets. The process of generating the buckets at different levels automatically adapts to the input order and data distribution. For the problem of wild values, we can store and maintain a set of smallest elements called the *lower set* and a set of largest elements called the *upper set*. The elements in these two sets are not inserted into any buckets. We name this new algorithm *dynamic bucketing* (DB). To be able to allocate space for the newly generated subbuckets, we have to reclaim the space wasted by the empty buckets resulting from skewed data distribution. Next, we will introduce some data structures to compress data histograms for better memory space utilization.

2.3 Data compression schemes

In SB, a CA is used to store the count values of the buckets. Since the CA for a sparse skewed dataset has many zeros, storing only the nonzero buckets could save space. Each nonzero bucket is represented by an (index, count) pair. We use an *array of pairs* (AP) to store the nonzero count values together with the indexes of the buckets. The equivalent AP of the CA for the running example is *(*(0, 27), (1,6), (2, 1), (6, 1), (16, 10), (17, 3), (19, 2)*)*, whose size is 14 vs. 20 for CA.

When there are many consecutive nonempty buckets, i.e., bursts of data, *array of ranges* (AR) compression can be used. A range has three parts (start index, end index, count array). The CA records the counts of the buckets between the two indexes. Choosing the right data structure requires a scan of the count array C in memory.

Define
Z = number of zeros in C, and
$S = \Sigma(t - 2)$.

The summation is over all consecutive nonempty bucket segments, and t is the length of the segment. For each segment we save the space for t indexes but incur space costs for starting and ending indexes for the range of the segment when comparing with AP.

If $Z > b - Z$, i.e., $Z > b/2$, use AP instead of CA. Here b is the total number of buckets in C.

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

Example: Suppose $b = 40$ and C = [0, 2, 0, 0, 0; 0, 3, 1, 2, 7 1; 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 $Z > b/2$ and $S > 0$, AP needs less memory than C and AR less than AP. The AR is $\langle(1,1,CA),(6,9,CA),(12,13,CA),(16,16,CA),(20,24,CA,(30,30,CA)\rangle$. The sizes of C, AP, and AR (in terms of the number of integers) are 40, 28, and 26, respectively.

If we can relax the “consecutive nonzero segments” condition to allow at most z consecutive zeros in a segment (z is usually small), AR could be further compressed. In our example, if $z = 2$, AR can be compressed into $\langle(1,1,CA),(6,16,CA),(20,24,CA),(30,30,CA)\rangle$. The size is still 26, but there is a smaller array (of ranges).

If a more aggressive scheme to save space is necessary, one can combine consecutive buckets with small histograms (e.g., smaller than m) together to form larger buckets whose histograms are the summations of the histograms of the buckets being combined. For example, if $m = 4$ and $C' =$ [1, 1, 0, 1, 2; 0, 0, 1, 6, 3; 0, 0, 2, 1, 0; 12, 1, 0, 2, 0; 2, 25, 3, 0, 1; 1, 1, 1, 0, 0; 1, 2, 0, 2, 3 1; 2, 3, 0, 2, 1], then C' is reduced to

$$\begin{pmatrix} [0,7] & [8,8] & [9,14] & [15,15] & [16,20] & [21,21] & [22,33] & [34,34] & [35,39] \\ 6 & 6 & 6 & 12 & 5 & 25 & 12 & 31 & 8 \end{pmatrix}.$$

The first row is the index ranges of combined buckets, and the second row is the count values of the ranges instead of CA. This scheme is more like equidepth histograms than equiwidth histograms.

Notice that, although AP and AR save space, they take more time than CA for insertions. We use balanced search trees (BSTs) to achieve faster insertions while saving space. For AP, a node consists of an (index, count) pair. If a red-black tree is used, the node size is at least 5 (index, count, two pointers, and color). For AR, a node consists of (start index, end index, count array). Because the ranges are disjoint, BST uses start indexes as keys.

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

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

Suppose δ_1, δ_2,..., δ_k are the step lengths of the sub-buckets for division and Δ is the step length of the first-level division. Then

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

3 Structure trees

3. 1 Data structure of structure trees

To represent the structure of the buckets that are on different levels of granularity and store the histograms, we pro-pose a new data structure called the *structure tree* (ST). An ST is a multiway and generally unbalanced tree. A node includes the following fields: range lower bound *low*, range upper bound *high*, number of buckets *num_buckets*, indexes of dense buckets *dense_index*, and data structure (DS) to store the counts (either CA or BST). The children of a node represent dense buckets and have finer granularity. Figure 4 shows an ST on the left, where the root node [0, 100,10; 0, 8; DS] indicates that the range [0, 100] is divided into ten buckets from which the 0th and the 8th buckets are dense and the DS is the CA.

```
1   Load initial elements into the lower set and upper set heaps;
2   Create the root using the tops of the two heaps as range bounds;
3   While ( not end of file) {
4       Read x from file;
5       IF ( x belongs to a heap) {
6           pop the heap top and insert it into the tree;
7           insert x into the heap; }
8       ELSE  insert(root, x);
9       IF (the number of records processed is a multiple of phase size) {
10          adjust each node in the tree and check if it is skewed;
11          create new nodes if necessary; }
12  }
13  insert(node, x) {
14      Compute index k = (x-low)*num_buckets/(high-low);
15      IF (x is in a sparse bucket) increase its count by 1 and return;
16      Otherwise find the corresponding child c of structure tree
17      Recursively insert into the child by calling insert(c, x);
18  }
```

Fig. 3. The initialization of structure trees

3.2 Initialization and maintenance of STs

We initialize the ST by scanning the input data set through phases. In each phase, a fixed number of elements are read and inserted. First, the heaps storing extreme values are initialized and the root of the ST is created (lines 1 and 2 in Fig. 3). While inserting an element *x*, first check if it belongs to a heap by comparing the root of the heap with *x*. If so, push *x* into the heap and switch out the top and insert it into the tree; otherwise, just insert *x* into the tree (lines 5 through 8). At the end of each phase, we adjust the tree nodes if they become skewed (lines 9–12). If the standard deviation of the histograms in a node exceeds a user-specified threshold, then the node is skewed and must spawn children. The bucket with the largest count (this count is set as the count threshold) is dense and will be further divided. Other buckets in the node will be deemed “dense” in later phases if their count values are larger than the count threshold. The bucket indexes of the new children are saved in the *dense_index* field. The bucket boundaries serve as range bounds for all the subbuckets.

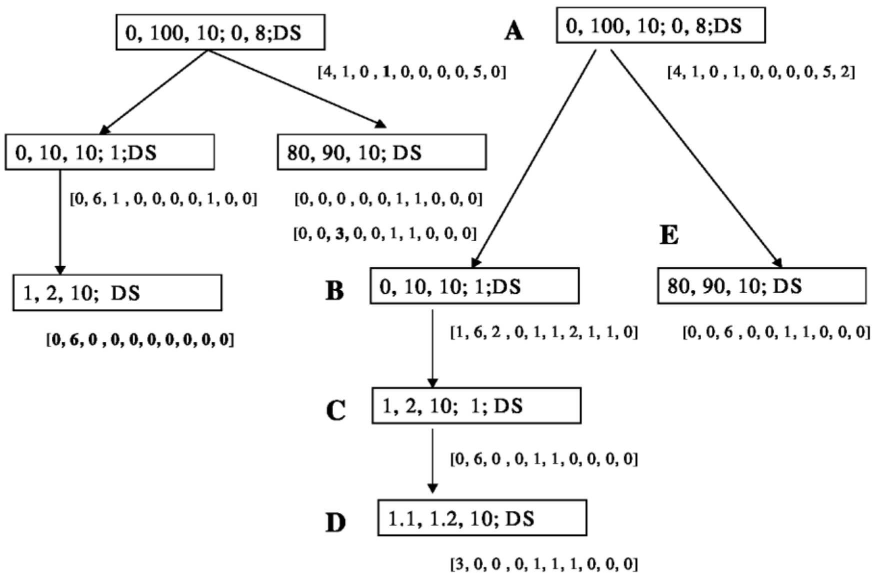


Fig. 4. The structure tree after inserting 30 elements and the final structure tree

The data structures for the DS histograms can also be adjusted at this time to save more space. For example, a CA can be replaced by a BST when (number of nonzero buckets) * BST_node_size < num_buckets. However, when (num nodes)* BST_node_size > num_buckets, the BST consumes more space and is slower and therefore needs to change back to a CA. An initial reserved space for the heaps is first taken out from the available memory space. The remaining space is maintained and a portion (say, half) of it is allocated to the dense buckets proportional to their count values. The conservative heuristic avoids allocation of too much space to one node and reserves some space for the insertions to BSTs. This memory administration ensures that at any time the total consumed space is within available limits. In the next phase, the histograms, including those of the newly generated subbuckets, are updated while new elements are inserted. Repeat this process until the end of the input file. In this way, the space allocation and tree structure adapt dynamically to the input data distribution and input order. The *insert* function is recursive. First, the bucket index of x is computed (line 14). If the element belongs to a sparse bucket, simply increase the count value of the matched bucket (line 15); otherwise, insert it recursively into a child that matches one of the dense indexes in the node (lines 16– 17). To simplify, we use a CA only in this figure. For BSTs, the count value of the matching sparse node in the BST increases by 1.

Next, we apply our new DB algorithm to the running example. Initially, we divide the whole range from 0 to 100 evenly into ten buckets and read ten elements from input data as the first phase (i.e., phase size is 10).

After insertion, the CA shown in the lower right part of the root of the left ST in Fig. 4 has two dense buckets (assume the count threshold here is 4): the 0th and 8th buckets. Therefore, indexes 0 and 8 are put in the indexes field of the root. Two new children are created and inserted into the ST. Suppose we fix *num_buckets* at 10 while generating subbuckets. In the second phase, the CAs of the new children are computed. After the third phase, the CAs are updated (in boldface fonts). The final tree is given in the right side of Fig. 4. Although this simplified example does not reflect all the aspects of the algorithm, it gives us an idea of how the algorithm works.

4 Query evaluation using dynamic bucketing and random bucketing

4. 1 Evaluation of selection queries

To find the i-th smallest element when $i < M$ or $i > N — M$ and M is the available memory size, the exact target can be found in only one pass regardless of the size of input data. This is achieved by using a max or min heap, which keeps a memory load of smallest or largest elements during the scan, as shown in lines 1–8 in Fig. 5. A “top n” or “bottom n” query can also be answered this way as long as the target is in the heaps. To answer multiple queries in general, first the ST is initialized as in Fig. 3. Starting from the root, a left-to-right scan of the buckets in a node while adding up the count values until the summation is larger than the target rank i. Recursively traverse the children of the dense buckets. Once the target bucket is located, its bucket boundaries serve as the lower and upper bounds of the search target. The summation of the matched bucket counts along the path from the target node to the root serves as the rank error guarantee.

Continuing the running example after the ST is initialized, we can traverse the tree in the right side of Fig. 4 in the order of A, B, C, D, C, B, A, E, A and sum up the counts up to 25 (i = 25, i.e., the median). Starting from the root A, since the first bucket is dense, we move to its child node B in the ST. The count sum is 1. We then move to C and D. At this point the sum of the counts is still 1, and then we visit all the buckets of D. The sum becomes 7(= 1+3+0+0+0+1+1+1+0+0+0). After revisiting C, the sum is 15 (= 7 + 6 + 1 + 1). Returning to node B, we stop at its fifth bucket since the sum reaches 25(15 + 6 + 2 + 0 + 1 + 1) there. So the target bucket is [5,6] and the estimate median is 5.5. To find rank error, we need to add up all the corresponding counts of the dense buckets along the path from the parent of the target node (i.e., B) to the root. In this example, the maximal rank error is 4.

4.2 Evaluation of rank queries and range queries

Rank queries and range queries can also be evaluated using the initialized ST in a similar fashion. In addition to solving the selection problem, the bucketing algorithm can also answer many other useful queries internally by exploiting the ST. For example, summing up the histograms of the buckets and subbuckets 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 by summing up the histograms of the buckets and subbuckets from the start bucket to the end bucket of the range bounds. Furthermore, the time cost can be amortized. The goal is to set up an internal data structure, i.e., ST, and then answer any number of queries in memory with high accuracy. Given the available memory space, the estimation is close to the true target in terms of both the value and the rank.

4.3 Clustering using dynamic bucketing

Clusters are dense buckets. There should be no clear clusters in uniformly distributed random datasets. Clustering, i.e., identifying the abnormal dense buckets, is useful for skewed datasets. Accurately finding the dense buckets in the ST after initialization is easy. Given a density threshold, we can identify the first-level clusters as follows. At the root, scan its bucket counts one by one and check if its histograms (plus the summation of the count values of all the buckets in the subtrees for the dense buckets) are larger than the threshold. If so, they are "clusters."

In our running example, the average frequency of the buckets at the root is 5(= 50/10). If we define the density of a bucket as the ratio of its histogram to its bucket length, then the average density of the root is 0.5(50/100). If the threshold is 10, only bucket 0 and bucket 8 are clusters. Their densities are 3.3(33/10) and 1.3(13/10), respectively. The two clusters include 92% of all the elements in the input data. The second-level clusters can be found similarly by examining all the buckets in the nodes of level 2. The average frequency is 50/100. For threshold 5, there are two clusters with density 20(20/1) and 6(6/1).

4.4 Randomized bucketing

Using random sampling techniques to choose the approximate quantiles as bucket boundaries is often a simple and effective way of solving the selection problem. In *randomized bucketing* (RB), choose *M* elements randomly from the input dataset in the first pass. One could use the sorted sample keys to de-fine 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 [7,8]. 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.

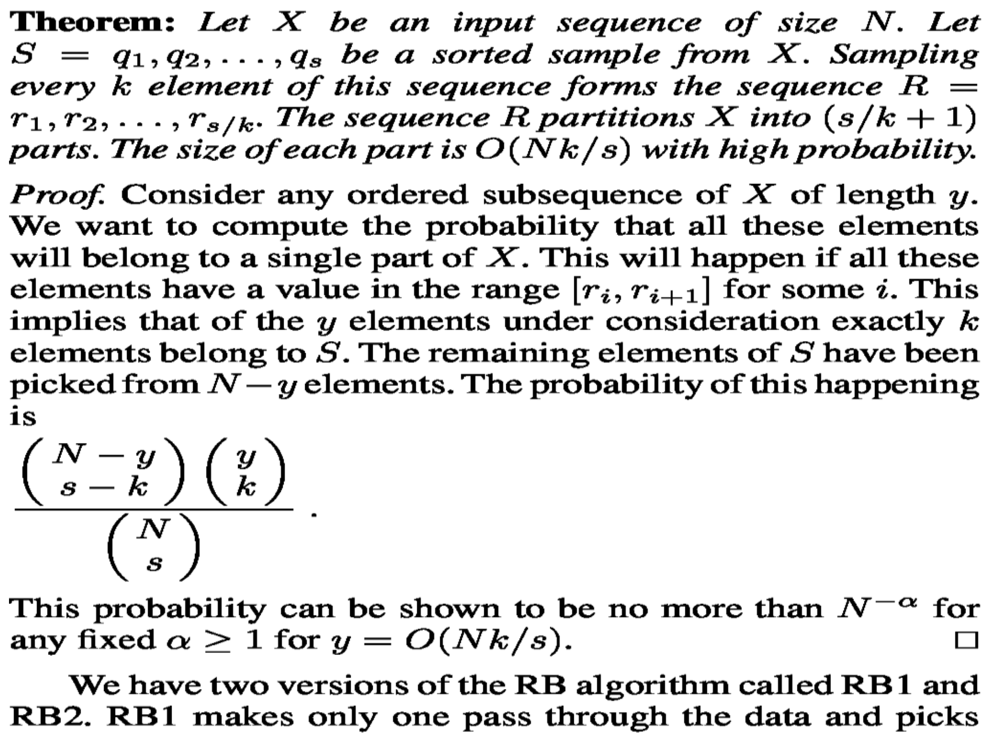
In fact, the above technique can be generalized to sample every *k*-th element of the sorted sample and use them to define bucket boundaries. In this case, we can show that with high probability the size of each bucket is $O(Nk/M)$.

```
If (i < M) { // Selection using a max heap
load first M elements from input data file and insert into a max heap H;
While (not end of input file) {
        read an input element x and compare with root element max of H;
        if ( x < max ) delete max from H and insert x into H; }
delete (M - i) elements from H;
Return root value of H as the final result;
Else if ( i > N-M ) solve the problem similarly using a min heap;

Initialize the structure tree using the algorithm in Fig. 3 ;
Traverse count arrays of the structure tree nodes starting from the root;
add up the count values in the buckets
        until the summation is greater than i;
If (a dense bucket is met)
        recursively traverse its child and sum up counts;
```

Fig. 5. Dynamic bucketing algorithm

```
create two empty buffers b0 and b1; j = 1;
While (more data in file) {
    load data into $b_0$ from input file;
    if ($b_1$ has room to store the new samples from b0)
        Transfer($b_0$, $b_1$);
    else {
        if ($b_j$ is full) create a new buffer $b_{j+1}$ on top; j = j+1;
        find the first non-full buffer bk upwards starting from b2;
        for i = k- 1 to 0 Transfer($b_i$, $b_{i+1}$);}
}
Compute $l$ and $u$ from top buffer;

Transfer(Lbuf, Hbuf) {
    compute p-samples of Lbuf;
    append the p-samples into Hbuf and then empty Lbuf; }
```

Fig. 6. ***Regular sampling algorithm***



Theorem: *Let X be an input sequence of size N. Let $S = q_1, q_2, \ldots, q_s$ be a sorted sample from X. Sampling every k element of this sequence forms the sequence $R = r_1, r_2, \ldots, r_{s/k}$. 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 of the y elements under consideration exactly k elements belong to S. The remaining elements of S have been picked from $N-y$ 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 $y = O(Nk/s)$. $\square$

We have two versions of the RB algorithm called 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 more pass through the data to compute the histograms of the buckets whose boundaries are the sample points. Finally, RB2 identifies elements that bracket the target and computes the selection element exactly. In our experiments, RB 1 was the fastest of all the algorithms tested. Its accuracy was not great but acceptable for roughly uniform data. RB2 took more time than RB 1 but produced much more accurate estimates. RB 1 is an excellent choice when one is interested in a quick estimate of quantiles. Note that the performance of RB 1 is independent of the input distribution and permutation.

5 Regular sampling and iterative sampling algorithms

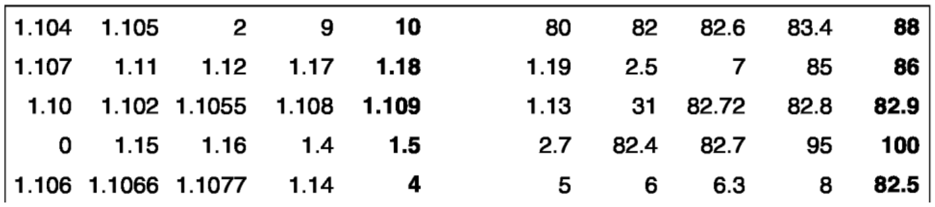
In this section, we will extend and adapt the top two comparison-based quantiling algorithms [12,16] to new selection algorithms.

5. 1 Regular sampling

The algorithm of Rajasekaran in [18] is an extension of the idea of Munro and Paterson [14]. A layer of $j + 1$ buffers $b_0, b_1, \ldots, b_j$ of the same size s holds the samples from the input set S. Value j depends on the sizes of s,

M, and N. The elements in buffer b_0 are read directly from input data S. The elements in b_1 are the samples of every other $\sqrt{M}$ element after b_0 is sorted (i.e., the exact $\sqrt{M}$-quantiles of b_0). We call these samples $\sqrt{M}$-samples. Similarly, buffer b_{k+1} is populated by the $\sqrt{M}$-samples from b_k, $k = 1, 2, \ldots, j — 1$. Bounds l and u are two elements from the top buffer b_j with rank is/Nd and $is/N + d$, respectively, where $d = j\sqrt{M}$.

Using a special fixed period $p = \sqrt{M}$ has a drawback of not having enough samples in the top buffer to compute ac-curate bounds l and u when N/M is reasonably large (e.g., 1 to $\sqrt{M}$). This often happens in practice. In this section, we generalize the above algorithm such that the sample period p can be any integer and extend the corresponding analysis. We name the generalized algorithm *regular sampling* (RS), whose pseudocode is shown in Fig. 6. Subroutine transfer computes the p-samples of Lbuf (i.e., sort Lbuf and then sample every p elements) and populates them into Hbuf. RS reads all input data and gets a stack of buffers by regular sampling layer by layer. If the top buffer is full, a new buffer should be created (line 7); otherwise, we can find the first nonfull buffer b_k upwards from b_2. A sequence of upward transfers is triggered to prepare more room for future new samples (lines 8 and 9). The ranks in the top buffer of the bounds R_l and R_u are computed (line 11): $R_l = i/p^j - d$ and $R_u = i/p^j + d$, where d is a rank relaxation parameter to pick the bounds. To ensure that the target is bracketed, we will prove later that d must be at least p.



1.104	1.105	2	9	**10**	80	82	82.6	83.4	**88**
1.107	1.11	1.12	1.17	**1.18**	1.19	2.5	7	85	**86**
1.10	1.102	1.1055	1.108	**1.109**	1.13	31	82.72	82.8	**82.9**
0	1.15	1.16	1.4	**1.5**	2.7	82.4	82.7	95	**100**
1.106	1.1066	1.1077	1.14	**4**	5	6	6.3	8	**82.5**

Fig. 7. The sorted buffers in regular sampling

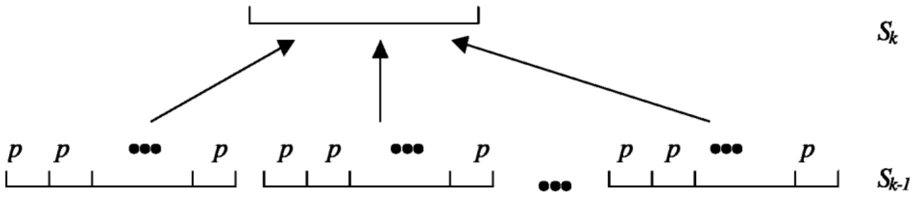


Fig. 8. Rank changes of a sample. Each buffer load contains s elements, and the size of S_{k-1} is sp^{j-k+1}

Applying regular sampling to our running example, we use two buffers b_0 and b_1 of size $s = 10$ each. We set the sampling period p to 5. Figure 7 shows the sorted buffer loads for b_0. The selected samples stored in b_1 are in bold fonts. The top buffer b_1 is sorted to: 1.109, 1.18, 1.5, 4, **10**, 82.5, 82.9, 86, 88, 100. Element 10 is the estimated median of the input data.

For the external selection problem, the lower bound of number of passes needed is two: in the first pass, lower bound*l* 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 of the parameters that lead to the optimality:

1. Bracketing: the target is contained between the bounds.
2. The number of survivors is no more than M.

Let us first introduce the following lemma.

Lemma *Let S_k be the set of all elements that have ever visited buffer b_k, $k = 0, 1, \ldots, j$. If the top buffer is full, then an element x with rank r in S_k has a rank in the range of* $(rp, rp + p^{j-k+1}(p - 1))$ *in S_{k-1}, $k = j, j - 1, \ldots,$* 1.

Proof. From the algorithm we know that $| S_k{-}1 | = p| S_k |$. If $|S_j | = s$, then $|S_{k-1} | = sp^{j-k+1}$. Since each sample in S_k represents p elements in S_{k-1}, the element x with rank r in S_k is larger than at least rp elements in S_{k-1}, i.e., the rank of x in S_{k-1} is at least rp. There are $| S_{k-1} | /s = p^{j-k+1}$ buffer loads of b_{k-1} (Fig. 8). In the worst case, at most $p - 1$ elements other than those previous smaller elements are smaller than x in each buffer load.[1] Otherwise, the rank of x in S_k would be larger than r. So the rank of x in S_{k-1} is atmost$rp+p^{j-k+1} (p- 1)$.

Theorem *Regular sampling will achieve the above optimality, if*

1. $d \geq j(p - 1)$,
2. $3jp^j (p - 1) < M$,
3. $N<M^2/ (3j (j + 1) (p - 1))$.

Proof. The estimated rank of the target in buffer j is i/p^j. If bounds l and u are picked to have ranks around this rank, then we may bracket the target in S_0. Let the elements l and u have rank $si/p^j - d$ and $i/p^j + d$ in S_j, respectively, where d is the rank relax parameter. Suppose the minimal and maximal ranks in S_{j-k} of the two bounds are $A^{(k)}$, $B^{(k)}$, $C^{(k)}$, and $D^{(k)}$, respectively (Fig. 9).

From the lemma we have

$$A^{(1)} = (i/p^j - d)p$$
$$B^{(1)} = (i/p^j - d)p + p(p - 1)$$
$$C^{(1)} = (i/p^j + d)p$$
$$D^{(1)} = (i/p^j + d)p + p(p - 1).$$

The minimal rank of$A^{(1)}$ in S_{j-2} is $A^{(2)} = pA^{(1)} = (i/p^j - d)p^2$, and the maximal rank of $B^{(1)}$ in S_{j-2} is $B^{(2)} = B^{(1)}p + p^{2}(p - 1) = (i/p^j - d)p^2 + 2p^2 (p - 1)$.

By repeatedly applying the lemma, we compute the minimal and maximal ranks of l in $S0A(j)$ and $B(j)$ as follows:

$$A^{(j)} = (i/p^j - d)p^j = i - dp^j$$
$$B^{(j)} = (i/p^j - d)p^j + jp^j (p - 1).$$

Similarly, we can compute the minimal and maximal ranks of u in S_0 $C^{(j)}$ and $D^{(j)}$:

$$C^{(j)} = i + dp^j$$

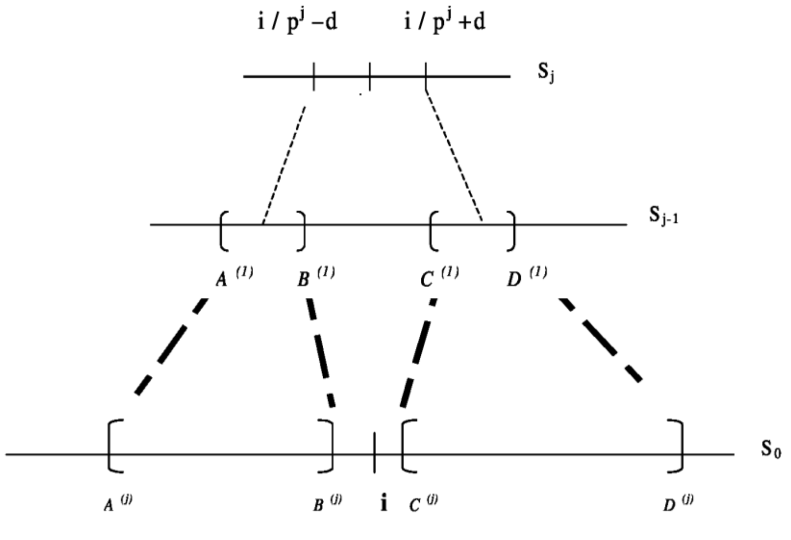


Fig. 9. Rank range of approximate target in S_0

$D^{(j)} = i + dp^j + jp^j\,(p - 1)$.

To bracket the target, $B^{(j)} \le i \le C^{(j)}$ must be satisfied. To ensure that all survivors after sifting fit into memory, $D^{(j)} - A^{(j)}$ must be less than M.

$B^{(j)} = i + jp^j(p - 1) - dp^j \le i \Rightarrow d \ge j(p - 1)$.

Equality ensures a more accurate estimation.

$D^{(j)} - A^{(j)} = jp^j\,(p - 1) + 2dp^j < M \Rightarrow 3jp^j\,(p - 1) < M$
$N = |S_0| = sp^j < sM/\,(3j\,(p - 1)) < M^2\,/(3j(j + 1)(p - 1))$

The last step incorporates the fact $(j + 1)\,s < M$. This completes the proof.

From the theorem, if $N > M^2$, in the worst case the RS algorithm will not be able to compute the true target internally after sifting. In fact, we have the following conjecture that not only can RS not do this, but no other selection algorithm can do it either in this case.

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

Rationale. To be able to solve the selection problem with arbitrary integer i, approximate p-quantiles are found with as many quantiles as they fit into memory. Suppose an algorithm finds such $(M - 1)$ values as the boundaries. All the input data are divided into M subsets. If $N > M^2$, according to the pigeonhole principle, there exists at least one subset whose size is larger than M. We can formulate a problem instance by putting the target in that subset so that algorithm cannot solve the selection problem internally. Notice that since data distribution and the input order are *unknown* in advance, we cannot really zero in on the target, even though we already know i.

5.2 Iterative sampling

Manku et al. [12] propose a uniform framework for selection that generalizes the Munro-Paterson algorithm [14] and the algorithm proposed in [3]. Manku et al.'s algorithm is characterized by two parameters b (number of buffers), s (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 out-put 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 set at 1. The level of the output buffer is one plus the level of the input buffers, and the weight of the output buffer is the summation of the input buffer weights. In COL-LAPSE operations, the input buffers with the same level value are sorted first. 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.[2] This COLLAPSE process is similar to merging in the external merge-sort algorithm. 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. In the resulting buffer of the OUTPUT operation, the approximate target position is at $[Wsi/N]$, where W is the summation of all COLLAPSE result buffers' weights.

The paper of Manku et al. also gives the upper bound of the rank difference between real target and output result.

*Rank_Error = (W-C-1)/*2 + *Wmax,* where

C: total number of COLLAPSE operations

Wmax: maximal weight of root’s children.

From the error guarantee we can compute the bounds that bracket the target, thus adapting the approximate quantiling algorithm to the selection problem. Define $q_1 = (i\text{-}Rank\ Error)/N$, $q_2 = (i+Rank\ Error)/N$, then the ranks of *l* and *u* in the final output buffer (include *W* copies for each element) are $[q_1\ sW]$ and $[q_2\ sW]$, respectively. These bounds are conservative as they are based on the guarantees. We term this adapted selection algorithm *iterative sampling* (IS). Its pseudocode is shown in Fig. 10.

```
1 While (not EOF){
2     NEW: fill available empty buffers (level=weight=1);
3         if (only one empty buffer has just been filled)
              level = the level value of top buffer of the stack;
4     push them into a stack;
5     pop top buffers with equal level value;
6     COLLAPSE them into output buffer;
7     empty the input buffers;
8     push the output buffer in stack;
9   }

10  OUTPUT the estimate from all the buffers in stack;
11  Compute l and u in the final output buffer if exact
        target is desired.
```

Fig. 10. Iterative sampling

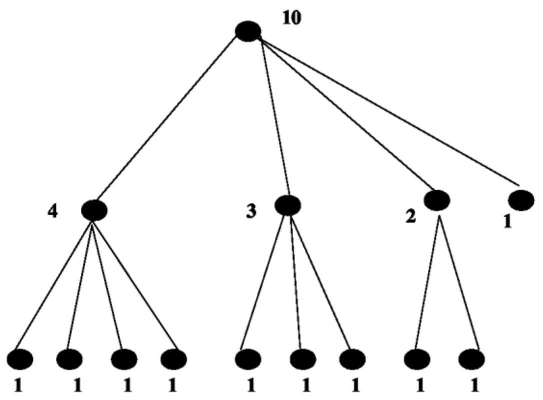


Fig. 11. A tree representation of operations for the running example

Continuing our running example, we use four buffers of size 5 each. The tree is shown in Fig. 11. The three divided boxes in Fig. 12 show the process of three COLLAPSE operations corresponding to the lower part of Fig. 11. To the left of the boxes are the sorted input buffers after NEW operations; to the right are realigned elements after sorting all the elements (with copies) in the input buffers. The elements in bold fonts are the contents of the output buffers whose weight and level values are also shown. Figure 13 shows the process of the final OUTPUT operation. From the final output buffer storing 1.102, 1.14, 2.5, 8, 82.9 we estimate the median of the input dataset to be 2.5.

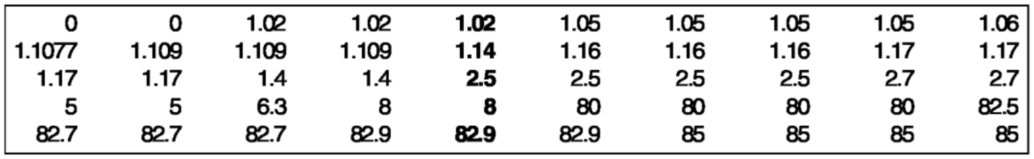
1.105	2	10	80	82	1.104	**1.105**	1.107	1.11
1.104	9	82.6	83.4	88	1.12	**1.17**	1.18	1.19
1.11	1.19	2.5	85	86	2	**2.5**	7	9
1.107	1.12	1.17	1.18	7	10	**80**	82	82.6
					83.4	**85**	86	88

level =2, weight=4

1.1	1.1055	1.109	1.13	31	1.1	**1.102**	1.105	level =2,
1.102	1.108	82.72	82.8	82.9	1.108	**1.109**	1.13	weight=3
1.15	1.16	82.4	82.7	95	1.15	**1.16**	31	
					82.4	**82.7**	82.72	
					82.8	**82.9**	95	

0	1.4	1.5	2.7	100		**0**	1.106	level =2,
1.106	4	5	6	8		**1.4**	1.5	weight=2
						2.7	4	
						5	6	
						8	100	

Fig. 12. Process of COLLAPSE operations



0	0	1.02	1.02	**1.02**	1.05	1.05	1.05	1.05	1.06
1.1077	1.109	1.109	1.109	**1.14**	1.16	1.16	1.16	1.17	1.17
1.17	1.17	1.4	1.4	**2.5**	2.5	2.5	2.5	2.7	2.7
5	5	6.3	8	**8**	80	80	80	80	82.5
82.7	82.7	82.7	82.9	**82.9**	82.9	85	85	85	85

Fig. 13. Process of OUTPUT operation

6 Simulation results

We have done extensive experiments to compare the performance of the algorithms discussed in this paper. In particular, our simulation results will show that the new bucketing algorithms have great advantages in terms of runtime and estimation accuracy. These experiments were done on a Dell Precision 330 with a 1.7-GHz CPU and Windows 2000 operating system.

6. 1 Datasets and overview of the simulations

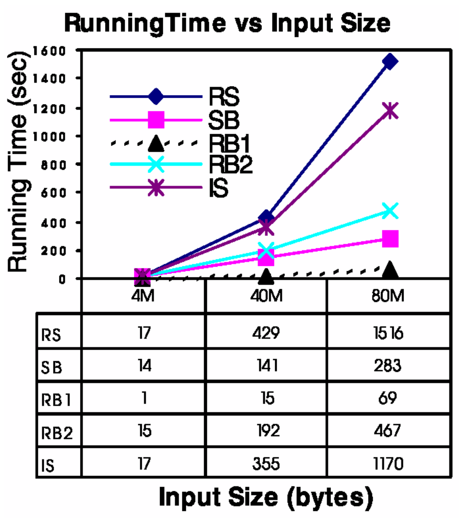
To comprehensively study the behavior and performance of the algorithms, we vary input data sizes, number of keys, data distributions, and the sizes of total allocated memory in different combinations. Data can be very skewed and have duplicates. Data input order is randomly shuffled. All these attempt to cover different cases for revealing the advantages and possible weaknesses of the algorithms studied. Using synthetic datasets in the simulations allows more control of the parameters of input data than real datasets might have. After generating the datasets, we wrote them as input disk files.

The parameters of six datasets D0 to D6 are shown in Table 1. For example, the first subset of D1 contains 4000 KB of data, which have 500,000 uniformly random keys in [0, 1]. We allocate the same amount of memory (80 KB) to all the algorithms studied. All the input data are shuffled to indicate the possible skewedness of input order. More details about the data will be given in the next two subsections.

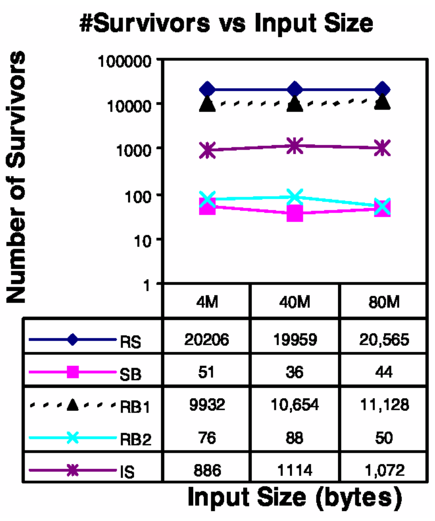
Table 1. Datasets used in the simulations

Datasets	size(KB)	num_keys(K)	mem_size(KB)	value range	distribution
	4,000	500	80		Uniformly
D1	40,000	5,000	800	[0, 1]	random
	80,000	10,000	1,600		
	4,000	500	80		Gaussian
D2	40,000	5,000	800	$[-0.1, 0.1]$	random
	80,000	10,000	1,600		$N(0.01, 0)$
	4,000	500	80		Uniformly
D3	4,000	500	400	[0, 1]	random
	4,000	500	800		
	4,000	500	40		
D4	4,000	500	80	[0, 1000]	Not skewed
	4,000	500	400		
	4,000	500	80		
D5	40,000	5,000	80	[0, 1000]	Not skewed
	80,000	10,000	80		
	4,000	500	80		
D6	40,000	5,000	80	[0, 10000]	Skewed
	80,000	10,000	80		

Datasets D_1 through D_3 are used in the next subsection to compare our bucketing algorithms with regular sampling (RS) and iterative sampling (IS). In Sect. 6.3, we compare DB with SB and P^2 using datasets D_4 through D_6. We compared the runtimes and accuracy of these algorithms under the same datasets with the same amount of memory. These algorithms were tested as single-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.

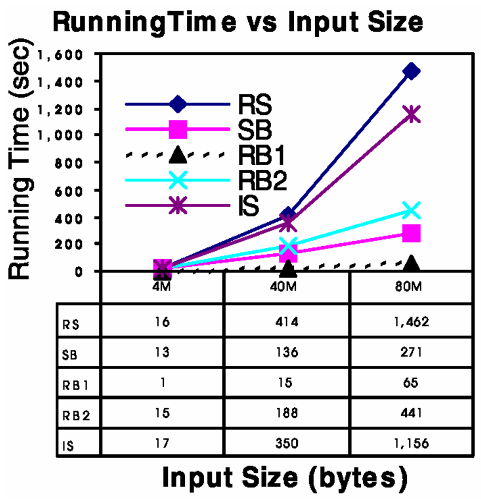


	4M	40M	80M
RS	17	429	1516
SB	14	141	283
RB1	1	15	69
RB2	15	192	467
IS	17	355	1170

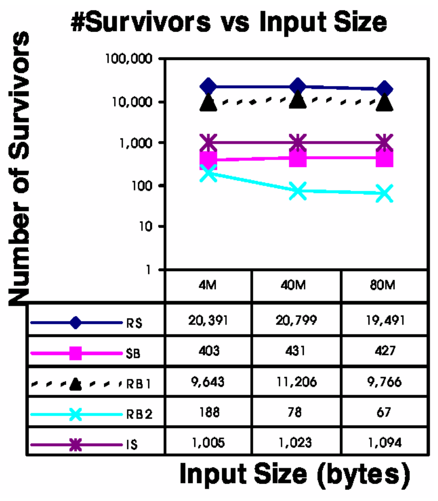


	4M	40M	80M
RS	20206	19959	20,565
SB	51	36	44
RB1	9932	10,654	11,128
RB2	76	88	50
IS	886	1114	1,072

Fig. 14. Comparison on dataset D_1, uniform random data in [0, 1]

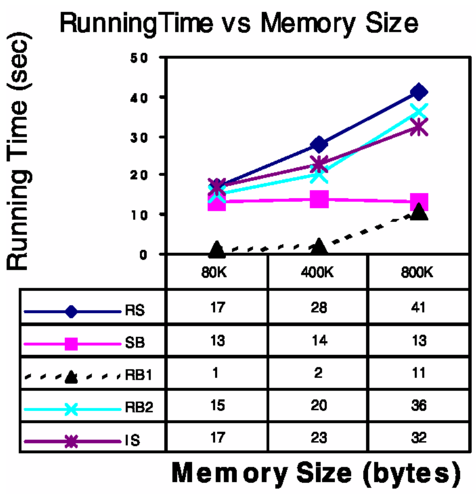


	4M	40M	80M
RS	16	414	1,462
SB	13	136	271
RB1	1	15	65
RB2	15	188	441
IS	17	350	1,156

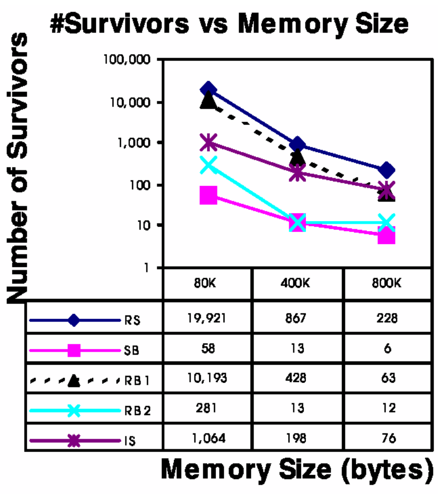


	4M	40M	80M
RS	20,391	20,799	19,491
SB	403	431	427
RB1	9,643	11,206	9,766
RB2	188	78	67
IS	1,005	1,023	1,094

Fig. 15. Comparison on dataset D_2: random data in normal distribution

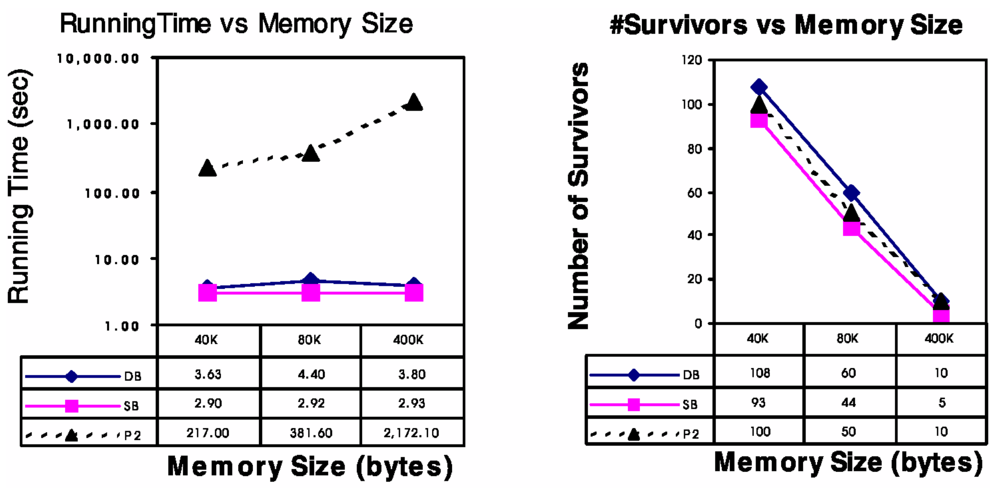


	80K	400K	800K
RS	17	28	41
SB	13	14	13
RB1	1	2	11
RB2	15	20	36
IS	17	23	32



	80K	400K	800K
RS	19,921	867	228
SB	58	13	6
RB1	10,193	428	63
RB2	281	13	12
IS	1,064	198	76

Fig. 16. Varying allocated memory sizes in D_3



	40K	80K	400K
DB	3.63	4.40	3.80
SB	2.90	2.92	2.93
P2	217.00	381.60	2,172.10

	40K	80K	400K
DB	108	60	10
SB	93	44	5
P2	100	50	10

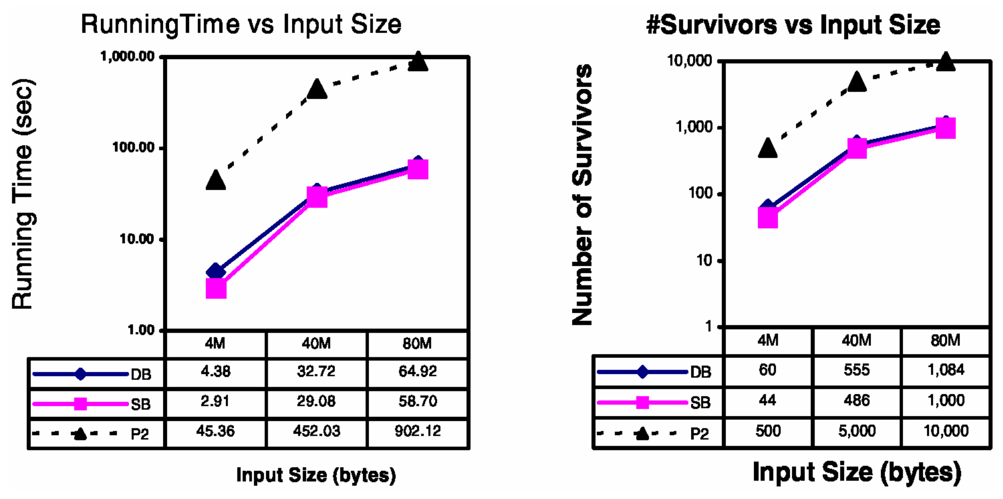
Fig. 17. Varying allocated memory sizes in D_4

6.2 Comparison with regular sampling and iterative sampling

In the first series of experiments using dataset D1, we explored the response times and accuracy of the algorithms with an increase of input data size or the number of keys. To indicate the scaling-up effect, the allocated memory sizes increase correspondingly so that the data size to memory size ratio is fixed at 50. We use a standard uniformly random generator, which generates keys in [0, 1]. The target element is the median.

We have also implemented a single-pass random sampling algorithm called RB 1, where the target is estimated directly by the random samples. We modify RB 1 into a two-pass algorithm called RB2 in which an additional pass computes the histograms using the random samples as bucket boundaries. We implement a two-level RS algorithm and use two buffers, each of size $M/2$. As in Sect. 2, let sampling period p be $2N/M$ and d be $(p - 1)$. In IS, we set $b = 7$. From Fig. 14 the bucketing schemes are faster than with RS or IS. RB2 could be slower when the dataset is large enough because it needs two passes. The single-pass randomized bucketing algorithm (RB 1) is by far faster than all the other algorithms, though its accuracy is not so good because the target is estimated merely by the random samples. When a fast and coarse estimation is called for, RB 1 is an excellent choice. The accuracy of SB and RB2 are better than the recent top algorithms RS and IS by 1 to 2 orders of magnitude.

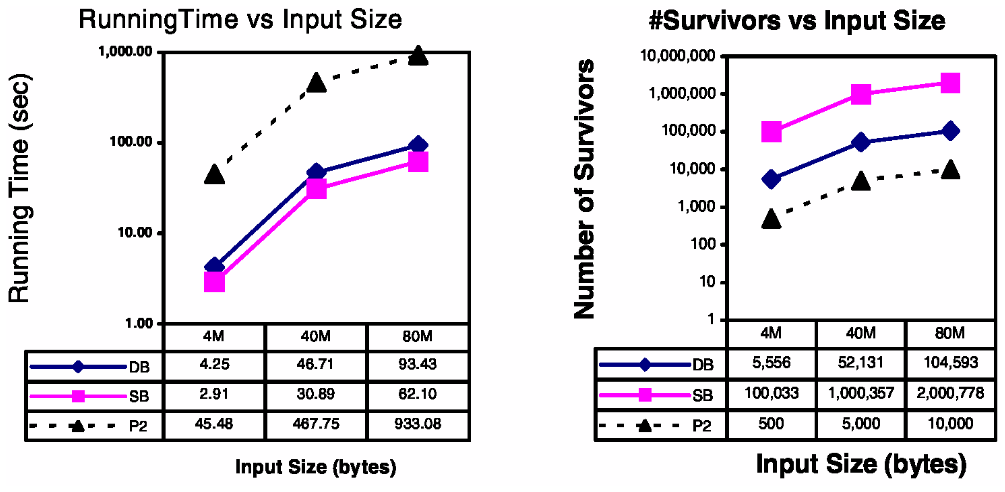
Dataset D_2 is designed for testing the behavior of algorithms under different data distributions. D_2 represents random data of normal distribution with variance 0.01 and mean value 0. We set the range bounds of SB to be −0. 1 and 0. 1. The results are shown in Fig. 15. A comparison of SB with uniform data reveals that the accuracy of SB deteriorates significantly while the behavior of other algorithms does not change significantly. Again, the bucketing algorithms have better performance.



	4M	40M	80M
DB	4.38	32.72	64.92
SB	2.91	29.08	58.70
P2	45.36	452.03	902.12

	4M	40M	80M
DB	60	555	1,084
SB	44	486	1,000
P2	500	5,000	10,000

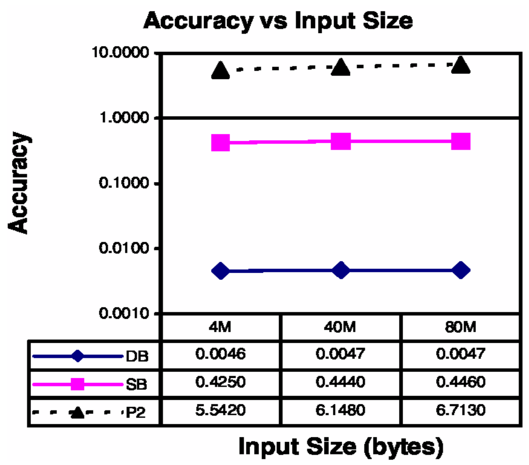
Fig. 18. Varying input size for nonskewed dataset D_5



	4M	40M	80M
DB	4.25	46.71	93.43
SB	2.91	30.89	62.10
P2	45.48	467.75	933.08

	4M	40M	80M
DB	5,556	52,131	104,593
SB	100,033	1,000,357	2,000,778
P2	500	5,000	10,000

Fig. 19. Varying input size for skewed dataset D_6



	4M	40M	80M
DB	0.0046	0.0047	0.0047
SB	0.4250	0.4440	0.4460
P2	5.5420	6.1480	6.7130

Fig. 20. Value accuracy comparison on D_6, skewed data

Dataset D_3 is used to investigate the scalability of the algorithms with respect to available memory sizes. Figure 16 shows that SB is more scalable in this sense while the others run slower with more memory because more data are processed each time in RS and IS (buckets are larger). In RB, a larger memory holds more samples. As expected, all algorithms improve their accuracy if more memory is allocated.

6.3 Comparison with simple bucketing and P^2

In this subsection, we first use D_4 to test the scalability of DB, SB, and P^2. Keys are uniformly distributed in [0, 1000]. In DB, we first obtain a random sample of 1% size of input data from which a standard deviation is calculated as the threshold and its minimum and maximum serve as the boundaries of the root. We set heap size at 50 and use 20 phases. Ninety percent of available memory is allocated for new dense buckets at the end of the phases. Figure 17 shows the results. Bucketing algorithms outperform P^2 by two orders of magnitude and are scalable with available memory size. Notice that DB is very close to SB, the ideal algorithm for uniformly distributed data. In fact, our new DB algorithm reduces to SB when the standard deviation and the available memory allocation ratio are large and fewer phases are used.

Figure 18 shows the results of running dataset D5, uniformly random keys in [0, 1000]. We vary the number of keys from 500,000 to 10 million but fixed the total allocated memory at 80KB. In view of the slowness of P^2, because of the search of the markers and the adjustments of the markers according to parabolic or linear curves, we use fewer markers (1000, one tenth of available memory) for P^2. With the increase of input data sizes, the running times increase and the accuracies decrease. DB and SB are about ten times faster and ten times more accurate than P^2.

Dataset D_6 is similar to D_5 but is skewed. Against the background of uniform data in [0, 10000], we add 20% more data clustered at 5000. The target is correspondingly adjusted to 60%. Figure 19 shows that, in comparison with nonskewed data, runtimes of all three algorithms do not change much but the accuracy of SB worsens severely (by four orders of magnitude worse). DB's accuracy has also degraded but is about 20 times more accurate than SB. P^2's accuracy is maintained. With similar precision, DB is, however, two orders faster. Furthermore, the value accuracy (true target minus the estimate) of DB is significantly better than SB and P^2 (Fig. 20).

7 Conclusions

In this paper, we have designed and analyzed two new quantiling and selection algorithms, namely, dynamic bucketing (DB) and randomized bucketing (RB). We extend two existing algorithms to new selection algorithms called regular sampling (RS) and iterative sampling (IS). When data are not too skewed, bucketing algorithms are faster and more accurate than these two algorithms. In particular, when MIN and MAX of input data are known and the data are approximately evenly distributed, SB is the best choice. However, when the data are very skewed and sparse, SB may incur unacceptable errors. In this case, DB retains the speed of SB with a high accuracy. Our extensive simulations on various datasets that vary data distribution, input sizes, memory sizes, etc. clearly show that our new algorithms outperform prior algorithms including RS, IS, and P^2 in terms of response times and estimation accuracy.

Notes:

1 To understand this, an analogy may be helpful. Imagine that there are p^{j-k+1} fishing strings (representing the buffer loads) each of which has s monotonous nondecreasing points and a knot every p points. The knot x with rank r is on the surface of the water. Other than the previous wet points, at most $p-1$ points on each string could be wet.

2 If there are two middle positions, we can choose the left one or randomly pick one.

References

1 . Agrawal R, Swami A (1995) A one-pass space-efficient algorithms for finding quantiles. In: Proceedings of the 7th inter-national conference on management of data (COMAD) Pune, India
2. Alsabti K, RankaS, SinghV (1997)A one-pass algorithm accurately estimating quantiles for disk-resident data. In: Proceedings of the 23rd conference on very large databases, Athens, Greece, 25–29 August 1997, pp 346–355
3. Blum M et al (1973) Time bounds for selection. J Comput Sys Sci 7:448–461
4. Feder T et al (2000) Computing the median with uncertainty. In: Proceedings of the 32nd annual ACM symposium on theory of computing, Portland, OR, May 2000, pp 602–607
5. Gray J et al (1997) Data Cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Mining Knowl Discov 1:29–53
6. Hellerstein JM, Haas PJ, Wang HJ (1997) Online aggregation. In: Proceedings of the ACM SIGMOD international conference on management of data, Tucson, AZ, May 1997, pp 171–182
7. Ho CT et al (1997a) Range queries in OLAP data cubes. ACM SIGMOD Rec 26(2):73–88
8. Ho CT, Bruck J, Agrawal R (1997b) Partial-sum queries in OLAP data cubes using covering codes. In: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, Tucson, AZ, May 1997, 228–237
9. Horowiz E, Sahni S, Rajasekaran S (1998) Computer algorithms. Freeman, San Francisco
10. Jain R, Chlamtac I (1985) The P^2 algorithm for dynamic calculation for quantiles and histograms without storing observations. CACM Commun Assoc Comput Mach 28:1076–1085
11. J'aJ'a J (1992) Parallel algorithms: design and analysis. Addison-Wesley, Reading, MA
12. Manku GS, Rajagopalan S, Lindsay BG (1998) Approximate medians and other quantiles in one pass and with limited memory. In: Proceedings of the ACM SIGMOD international conference on management of data, Seattle, June 1998. ACM Press, New York
13. Motwani R, Raghavan P (1995) Randomized algorithms. Cam-bridge University Press

14. Munro JI, Paterson MS (1980) Selection and sorting with limited storage. Theor Comput Sci 12:315–323
15. Piatetsky-Shapiro G (1984) Accurate estimation of the number of tuples satisfying a condition. In: Proceedings of ACM SIG-MOD, Boston, June 1984, pp 256–275
16. Poosala V et al (1996) Improved histograms for selectivity estimation of range predicates. In: Proceedings of ACM SIGMOD, Montreal, 4–6 June 1996, pp 294–305
17. Rajasekaran S (1995) Sorting and selection on interconnection networks. DIMACS series in discrete mathematics and theoretical computer science, vol 21. American Mathematical Society Publications, Providence, RI, pp 275–296
18. Rajasekaran S (2001) Selection algorithms for parallel disk systems. J Parallel Distrib Comput 61(4):536–544
19. Rajasekaran S, Reif JH (1993) Derivation of randomized algorithms for sorting and selection. In: Paige RA, Reif JH, Wachter RF (eds) Parallel algorithms derivation and program transformation. Kluwer international series in engineering and computer science, vol 23 1. Kluwer, Amsterdam, pp 187–205
20. Reif JH, Valiant LG (1987) A logarithmic time sort for linear size networks. J ACM 34(1):60–76
21. Vitter JS, Shriver EAM (1994) Algorithms for parallel memory. I: Two-level memories. Algorithmica 12:110–147
22. Wu YL, Agrawal D, Abbadi AE (2000) Using wavelet decomposition to support progressive and approximate range-sum queries over data cubes. In: Proceedings of the 9th international conference on information knowledge management (CIKM), McLean, VA, 6–11 November 2000, pp 414–421