A One-Pass Algorithm for Accurately Estimating Quantiles for Disk-Resident Data*

Khaled Alsabti
School of CIS
Syracuse University
kaalsabt@top.cis.syr.edu

Sanjay Ranka
Department of CISE
University of Florida
ranka@cise.ufl.edu

Vineet Singh
Information Technology Group
Hitachi America, Ltd.
vsingh@hitachi.com

Abstract

The φ-quantile of an ordered sequence of data values is the element with rank φn, where n is the total number of values. Accurate estimates of quantiles are required for the solution of many practical problems. In this paper, we present a new algorithm for estimating the quantile values for disk-resident data. Our algorithm has the following characteristics: (1) It requires only one pass over the data; (2) It is deterministic; (3) It produces good lower and upper bounds of the true values of the quantiles; (4) It requires no a priori knowledge of the distribution of the data set; (5) It has a scalable parallel formulation; (6) Extra time and memory for computing additional quantiles (beyond the first one) are constant per quantile.

We present experimental results on the IBM SP-2. The experimental results show that the algorithm is indeed robust and does not depend on the distribution of the data sets.

1 Introduction

The φ-quantile of an ordered sequence of data values is the element with rank $\varphi \times n$, where n is the total number of values. The median of a set of data is the 0.5-quantile. Quantiles or accurate estimates of quantiles are required for the solution of many practical problems.

Query optimizers need accurate estimates of the number of tuples satisfying various predicates. Methods were proposed in [PS84] to use quantiles for this purpose. Also, quantile algorithms can generate equi-depth histograms [PIHS96], which have been used to estimate query result sizes. In the past, equi-depth histograms [Koo80, PS84, MD88] have not worked well for range queries when data distribution skew has been high. Our new algorithm called OPAQ (for One Pass Algorithm for Quantiles; pronounced opaque) promises better results due to its combination of accuracy and efficiency features.

Quantiles can be used for computing association rules for data mining as shown in [AS95, AIS93, AS96]. Also, quantiles can be used for external sorting. Data can be partitioned using quantiles into a number of partitions such that each partition fits into main memory. Further, quantiles are excellent for load balancing many parallel applications [D+91].

The problem of finding a φ-quantile of a set of elements of size n which reside in the main memory can be solved in $O(n)$ time by using the deterministic algorithm of [B+72] or in $O(n)$ expected time by using the randomized algorithm of [FR75].

In many cases, the exact value of the quantile is not needed and a good estimate of the true value is sufficient. In this paper, we present an algorithm for estimating the φ-quantile ($\varphi = \frac{1}{q}, \frac{2}{q}, \ldots, \frac{q-1}{q}$) for large data sets. We assume that the data size is larger than size of the memory and the data is disk-resident.

Algorithms for estimating quantiles can be classified based on the following characteristics:

- *Number of passes (single/multiple)*: The number of passes of the input data required.

*A large part of this work was done while Khaled Alsabti and Vineet Singh were at IBM TJ Watson Research Center.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

- *Determinism*: The running time of the algorithm can be deterministic or randomized.
- *Accuracy*: This represents the lower and upper bounds on the error from the true value. Some algorithms provide probabilistic bounds only.
- *Data distribution*: Some algorithms can provide a good estimate only for certain data distributions.
- *Parallelization properties*: This represents the parallelization properties of the algorithm.
- *Cost for finding additional quantiles*: In some cases, additional quantiles may be required. This represents the cost of finding additional quantiles.

A one pass algorithm for estimating quantiles has been proposed in [AS95]. One limitation of this algorithm is that it does not provide an upper bound of the error rate. The algorithm partitions the range of the values into k intervals and counts the values in each interval. The boundaries of intervals are determined on-the-fly and are continuously adjusted as data is read from disk. A technique that needs multiple passes over the data and produces accurate quantiles was proposed in [GS90]. This algorithm uses a linear median-finding algorithm recursively to partition the data. An algorithm based on sampling [Coc77] and the algorithm proposed in [SD77] both require a priori knowledge of the data set in order to produce a good estimate of the quantile. The sampling algorithm works as follows. Draw a random subset of the data set as a sample. Then, sort the sample and use it to estimate the quantile values. In [SD77], an algorithm was proposed which partitions the range of values into k intervals. The algorithm counts the number of elements in each interval. The counts of the intervals are used to estimate the quantile value. Unless we have a priori knowledge of the data distribution, this algorithm may produce inaccurate estimates for quantile values. An algorithm which does not require a priori knowledge of the data distribution and requires one pass over the data was proposed in [JC85]. In this algorithm, they store a constant number of elements and update the elements as the input data is read. This algorithm does not provide any error bounds for the quantile estimates. In [MP80], single pass and multipass algorithms were proposed. The single pass algorithm produces an accurate quantile and requires $O(n)$ amount of main memory, where n is the total number of elements.

In this paper, we present a new algorithm OPAQ for estimating the quantiles. The OPAQ algorithm has the following characteristics:

- It requires only one pass over the data.
- It is deterministic.
- It produces tight lower and upper bounds of the true value of the φ-quantile.
- It requires no a priori knowledge of the data distribution.
- It has a scalable parallel formulation.
- The additional time and space complexity for estimating each additional quantile beyond the first one is constant per quantile.

The rest of this paper is organized as follows. Section 2 describes and analyzes the sequential algorithm. We also present experimental results in this section. Sections 3 presents the parallel algorithm. In this section, we also describe the parallel machine model and present the experimental results on the IBM SP-2. We conclude in section 4.

2 The Sequential Algorithm

In this section, we present a new algorithm for estimating φ-quantiles. In order to describe the new algorithm, we need to define a few terms. These terms are defined in Table 1.

Table 1: The definition of the terms

Term	Description
M	size of the main memory
m	size of each run
n	total number of elements
r	number of runs ($= \frac{n}{m}$)
s	size of the sample for each run
φ	quantile fraction ($\varphi \in [0 \ldots 1]$)
α	index (rank) of the quantile ($= \varphi \times n$)
e_α	value of the quantile

The OPAQ algorithm consists of two phases: sampling phase and the quantile finding phase. In the sampling phase, we input the whole data set as r runs. A set of sample points $S = [s_1, \ldots, s_s]$ of size s is determined where $s_i <= s_{i+1}$, for $i < s$, for each run. The r sample lists are merged together forming a sorted sample list of size rs. The sorted sample list is used in the quantile finding phase to estimate the upper and lower bounds of the true value of φ-quantile. The accuracy of the result depends on both the phases. These two phases are described in the next subsections.

2.1 The Sample Phase

Figure 1 gives a high level description of the sampling phase. To estimate e_α, we obtain an upper bound ${e_\alpha}^u$ and a lower bound ${e_\alpha}^l$ such that $e_\alpha \in [{e_\alpha}^l, {e_\alpha}^u]$

and the number of elements in the interval $[e_\alpha{}^l, e_\alpha{}^u]$ is bounded. The samples are selected using regular sampling [LLS+93]; a sample of size s consists of the elements at relative indices $\frac{m}{s}, \ldots, s\frac{m}{s}$.[1] Each sample point thus corresponds to $\frac{m}{s}$ points less than or equal to the sample point and greater than the previous sample point. We will use the term sub-run of the sample point to denote these elements.

The problem of finding a sample point at index k is exactly the same as finding the k^{th} smallest element in the given run. The problem of finding the k^{th} smallest element in a set of data is known as the *selection* problem. Many algorithms have been proposed to solve the selection problem. Some of these algorithms are deterministic and others are probabilistic. A deterministic algorithm is proposed in [B+72] with $O(m)$ worst-case running time, where m is the number of elements. A randomized algorithm has been proposed in [FR75] with expected and worst case times of $O(m)$ and $O(m^2)$ respectively.

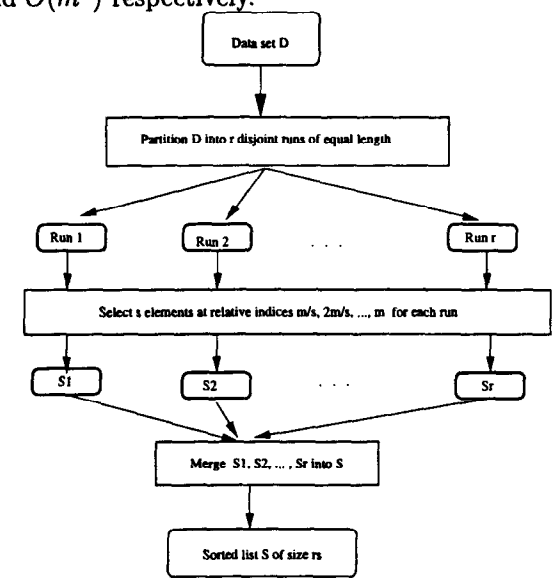

Figure 1: High level description of the sample phase. The data set D is of size n. Each run is of size m. s sample points are derived from each run.

The s sample points can be found as follows.[2] First, find the median of the m elements and divide the list into two equal parts. Then, find the medians of the two new sublists, and so on until the sizes of the sublists reach $\frac{m}{s}$. The sizes of the sublists will be $\frac{m}{s}$ after $\log s$ iterations. After $\log s$ iterations, we will have s sublists each of size $\frac{m}{s}$. The maximum element of sublist i is the i^{th} sample point and it can be found in time $O(\frac{m}{s})$. Using results in [B+72], we can find the s sample points in $O(m \log s)$ worst-case running time. The randomized algorithm for selection can be extended to find the s sample points in $O(m \log s)$ expected time and $O(m^2)$ worst case time. This algorithm has a small constant factor and is practically very efficient. After finding the r sample lists, we merge them together to form one sorted sample list of size rs.

[1] Without loss of generality, we assume that n is divisible by m and m is divisible by s.

[2] Assume that s and m are powers of 2. If they are not, it is easy to modify the algorithm slightly and maintain the same complexity.

2.2 The Quantile Phase

In this phase, we find $e_\alpha{}^l$ and $e_\alpha{}^u$ using the sorted sample list. As a result of using regular sampling method in deriving the sample points, it can be easily shown that the sample points have the following properties:

1. There are at least $i\frac{m}{s}$ elements less than or equal to the sample point s_i.
2. Additionally, there are at most $r-1$ sub-runs each with at most $\frac{m}{s}-1$ elements less than s_i.

Thus the maximum number of elements less than s_i is given by $i\frac{m}{s} + (r-1)(\frac{m}{s}-1)$. These properties are used in determining $e_\alpha{}^l$ and $e_\alpha{}^u$. For more details see appendix A.

Let *List* be the list of sorted samples. We assign $e_\alpha{}^l$ to be the i^{th} element in the sorted samples list such that:

$$i\frac{m}{s} + (r-1)(\frac{m}{s}-1) \leq \alpha < (i+1)\frac{m}{s} + (r-1)(\frac{m}{s}-1) \quad (1)$$

Solving formula (1) for i, we get

$$i = \lfloor \frac{s}{m}\alpha - (r-1)(1-\frac{s}{m}) \rfloor \quad (2)$$

This corresponds to

$$e_\alpha{}^l = List[\lfloor \frac{s}{m}\alpha - (r-1)(1-\frac{s}{m}) \rfloor] \quad (3)$$

Similarly $e_\alpha{}^u$ is the j^{th} element in the sorted samples list such that:

$$(j-1)\frac{m}{s} < \alpha \leq j\frac{m}{s} \quad (4)$$

This corresponds to

$$e_\alpha{}^u = List[\lceil \alpha\frac{s}{m} \rceil] \quad (5)$$

Lemma 1 *The maximum number of elements between the true quantile and the lower bound $e_\alpha{}^l$ is $\frac{n}{s}$.*

Proof: Let N_L be the maximum number of elements between $e_\alpha{}^l$ and the true value of the quantile, and $N_{min}(Cond)$ be the minimum number of elements which satisfy the condition *Cond*. Thus,

$$N_L \leq \alpha - N_{min}(Elements < e_\alpha{}^l)$$
$$= \alpha - i\frac{m}{s}$$

Substituting the value of i from formula (2), we get

$$N_L = \alpha - \lfloor \frac{s}{m}\alpha - (r-1)(1-\frac{s}{m}) \rfloor \frac{m}{s}$$

Thus,

$$N_L \leq \alpha - (\frac{s}{m}\alpha - (r-1)(1-\frac{s}{m}) - 1)\frac{m}{s}$$
$$\Longrightarrow$$
$$N_L \leq \frac{n}{s} - r + 1$$

Thus, the maximum number of elements between the true quantile and the lower bound ${e_\alpha}^l$ is at most $\frac{n}{s}$. ■

Lemma 2 *The maximum number of elements between ${e_\alpha}^u$ and the true quantile is $\frac{n}{s}$.*

Proof: Similar to lemma 1. ■

Lemma 3 *The maximum number of elements between ${e_\alpha}^l$ and ${e_\alpha}^u \leq 2\frac{n}{s}$.*

Proof: Straightforward from lemmas 1 and 2. ■

2.3 Time requirements

Table 2 summarizes the time requirements of the different steps. The total time required for estimating q quantiles is $O(n + rm\log s + rs\log r + q)$. This simplifies to $O(n + n\log s + \frac{n}{m}s\log\frac{n}{m} + q)$, since $r = \frac{n}{m}$. If $\frac{m\log s}{s} \geq \log\frac{n}{m}$, the total complexity of the algorithm is $O(n\log s)$. The size of the main memory M, the size of the sample s, the number of runs r and the number of elements n are constrained by the following relation:

$$rs + \frac{n}{r} \leq M$$

Since $s \geq 2q$ for achieving good bounds on the quantiles, this limits the maximum number of quantiles one can find using our algorithm to $O(\frac{M^2}{n})$.

An additional advantage of our algorithm is that the sample phase does not depend on the quantile phase. The same sorted sample list can potentially be used for finding other quantiles.

Table 2: The time requirement of the different parts of the algorithm

Phase	Complexity
Reading From the Disk	$O(n)$
Finding the rs sample points	$O(rm\log s)$
Merging r sample lists	$O(rs\log r)$
Estimating q quantiles	$O(q)$
Total	$O(n + n\log s + rs\log r + q)$

2.4 Experimental Results

We have conducted several experiments to evaluate our algorithm on a variety of data sets and compared performance with other algorithms presented in the literature. Our choice of particular data sets in terms of size and distribution of the keys reflects choices made in the literature for ease of comparison.

We conducted three experiments with data set sizes of 1 million, 5 million, and 10 million. For each data set size, the generated keys are chosen from either a uniform distribution or Zipf distribution [Zip49]. Further, the number of duplicates for each data set of size n is set to $\frac{n}{10}$. This was done to study the impact of data distribution on the accuracy of the estimates obtained. From the discussions in the previous sections, it is easy to observe that the time requirements of our algorithm are relatively independent of the underlying data distribution.

The Zipf distribution has a parameter which determines the degree of the skew of the data. The data set corresponds to a uniform distribution when the parameter is set to one. The level of skew increases as the value of this parameter decreases. The data set will have a very high degree of skew for the parameter value equal to zero. We chose 0.86 as the Zipf distribution parameter. Again, this reflects our desire to compare the performance of our algorithm to previously proposed algorithms.

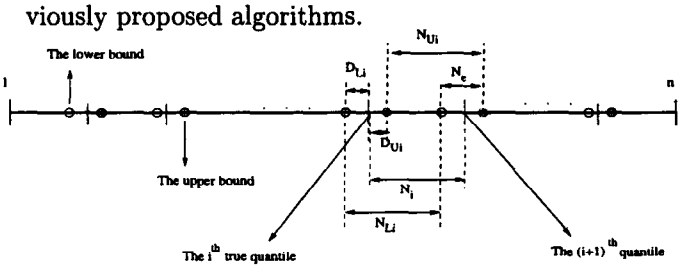

Figure 2: The definitions of the terms which are used in the relative error rates

The errors in estimating quantiles using our algorithm can be quantified using several measures. In this paper, we use three measures of error:

1. $RER_A = (N_e - N_t)/n \times 100$
2. $RER_L = Max_{i=1}^{q}(Max(\frac{|N_i - N_{Li}|}{N_i}, \frac{|N_i - N_{Ui}|}{N_i})) \times 100$
3. $RER_N = Max_{i=1}^{q}(Max(\frac{D_{Li}}{\frac{n}{q}}, \frac{D_{Ui}}{\frac{n}{q}})) \times 100$

The terms used in the different error rates are explained in Figure 2. Elements from the data set are shown in the figure in increasing sorted order from left to right. N_e is the number of elements between the estimated lower and upper bound. N_t (not shown in figure) is the number of duplicates for the exact quantile value between these bounds. N_i is the number of elements between the true i^{th} quantile and $(i+1)^{th}$ quantile, N_{Li} is the number of elements between the estimated lower bounds of the i^{th} and $(i+1)^{th}$ quantiles. N_{Ui} is defined similarly for the upper bound. D_{Li} is the number of elements between the true i^{th}

quantile and the lower bound of the i^{th} quantile. D_{Ui} is defined similarly for the upper bound of the quantile.

RER_A (A for Almaden) is taken from [AS95]. Note that this error rate is expressed in terms of the size of the whole data set. RER_L (L for Load Balancing) is useful for determining the difference in the positions of successive quantiles. This is useful for load balancing on a parallel computer. RER_N (N for Normalized) is a normalized error rate and does not depend on the total data size. Instead, the denominator is the number of elements between consecutive (actual) quantiles.

Table 3: The RER_A produced by the OPAQ algorithm for different sample sizes for data sets of size 1 Million

Decile	Uniform Distribution			Zipf Distribution		
	$s =$ 250	$s =$ 500	$s =$ 1000	$s =$ 250	$s =$ 500	$s =$ 1000
10%	0.33	0.17	0.08	0.33	0.12	0.08
20%	0.39	0.17	0.09	0.35	0.13	0.09
30%	0.39	0.17	0.09	0.34	0.18	0.09
40%	0.27	0.20	0.07	0.29	0.17	0.09
50%	0.38	0.18	0.09	0.30	0.16	0.07
60%	0.39	0.17	0.08	0.39	0.19	0.07
70%	0.37	0.17	0.10	0.36	0.18	0.06
80%	0.36	0.17	0.09	0.37	0.15	0.08
90%	0.35	0.19	0.08	0.15	0.15	0.09

For each data set, we report RER_A, RER_L and RER_N. Based on the lemmas 1 through 3, it can be easily shown that the upper bounds of RER_A, RER_L, and RER_N produced by the OPAQ algorithm are $\frac{2}{s} \times 100$, $\frac{q}{s} \times 100$, and $\frac{q}{s} \times 100$ respectively. Thus, the accuracy of the estimated value of the quantiles is directly proportional to the sample size.[3]

We obtained these error rates for different sample sizes for finding dectiles (i.e., 10%,20%,...,90%) of 1 million elements. Tables 3 and 4 show the relative error rates produced by the OPAQ algorithm for different values of s; the size of each run was equal to 100,000 elements. As expected, doubling the value of s results in approximately half the amount of the error. Although the execution time is not presented here, we observed that as the sample size s increases, the cost of finding the sample points and merging r sample lists gets larger.

The error rates for the OPAQ algorithm for finding dectiles of 1 million, 5 million, and 10 million data sets are shown in Tables 5 and 6. The sample size s and the run size m are set to 1,000 and 100,000 respectively. The results show that the accuracy of the algorithm does not significantly depend on the distribution of the data set.

We have also compared the accuracy of the OPAQ algorithm with random sampling and the algorithm proposed in [AS95] for RER_A; these results are presented in [AS95]. Assuming that the same amount of memory is provided to each of the three algorithms for their samples or data structures,[4] we found that the RER_A produced by our algorithm is comparable or better than the other two algorithms. However, the main strength of our algorithm is that we can bound the error for a given sample size. Table 7 shows the RER_A for data sets of size 1 million.

[3]The sample size is clearly limited by the amount of memory available.

Table 4: The RER_L and RER_N produced by the OPAQ algorithm for different sample sizes for data sets of size 1 Million

Decile	Uniform Distribution			Zipf Distribution		
	$s =$ 250	$s =$ 500	$s =$ 1000	$s =$ 250	$s =$ 500	$s =$ 1000
RER_L	1.88	0.99	0.46	1.88	0.89	0.52
RER_N	2.62	1.15	0.60	2.68	1.09	0.53

Table 5: The RER_A produced by the OPAQ algorithm for different data sizes

Decile	Uniform Distribution			Zipf Distribution		
	1M	5M	10M	1M	5M	10M
10%	0.08	0.09	0.09	0.08	0.09	0.10
20%	0.09	0.09	0.09	0.09	0.09	0.09
30%	0.09	0.09	0.10	0.09	0.09	0.10
40%	0.07	0.10	0.09	0.09	0.09	0.09
50%	0.09	0.09	0.09	0.07	0.10	0.09
60%	0.08	0.09	0.10	0.07	0.09	0.09
70%	0.10	0.09	0.10	0.06	0.09	0.09
80%	0.09	0.09	0.09	0.08	0.09	0.09
90%	0.08	0.09	0.10	0.09	0.10	0.09

3 Parallel Algorithm

Evolutionary trends of parallel computers have converged to a general architecture which consists of a small set (ten to a few thousand) of processing elements connected through an interconnection network. These coarse grained parallel machines have memory physically distributed across the processors. Interprocessor communication is either through message passing or through a shared address space. In this section, we describe the parallelization of our algorithm on such machines.

Rather than making specific assumptions about the underlying network, we assume a two-level model of computation. The two-level model assumes a fixed

[4]This corresponds to 3000 sample points (rs) in the OPAQ algorithm. The sample size s and the run size m are set to 600 and 200,000, respectively.

Table 6: The RER_L and RER_N produced by the OPAQ algorithm for different data sizes

Dectile	Uniform Distribution			Zipf Distribution		
	1M	5M	10M	1M	5M	10M
RER_L	0.46	0.51	0.53	0.52	0.53	0.54
RER_N	0.60	0.58	0.55	0.53	0.54	0.54

Table 7: Comparisons with the other two algorithms. Alg. 1 is proposed in [AS95] and Alg. 2 is the random sampling algorithm

Dec-tile	Uniform Distribution			Zipf Distribution		
	OPAQ	Alg. 1	Alg. 2	OPAQ	Alg. 1	Alg. 2
10%	0.13	0.4	0.1	0.12	0.0	0.1
20%	0.15	0.4	0.3	0.14	0.0	0.2
30%	0.15	0.1	0.5	0.15	0.1	0.4
40%	0.13	0.6	0.5	0.10	0.4	0.1
50%	0.13	0.5	0.5	0.12	0.5	0.1
60%	0.15	0.5	0.0	0.16	0.4	0.1
70%	0.16	0.3	0.1	0.15	0.1	0.3
80%	0.13	0.0	0.1	0.16	0.2	0.0
90%	0.14	0.1	0.2	0.07	0.3	0.0

cost for an off-processor access independent of the distance between the communicating processors. A unit computation local to a processor has a cost of δ. Communication between processors has a start-up overhead of τ, while the data transfer rate is $1/\mu$. For our complexity analysis we assume that τ and μ are constant, independent of the link congestion and the distance between two nodes. This permits us to use the two-level model and view the underlying interconnection network as a virtual crossbar network connecting the processors. It closely models the interconnection network on the IBM SP-2 on which we will present our experimental results. Although our algorithm is analyzed under the assumptions of a virtual crossbar, it is relatively architecture-independent and can be efficiently implemented on other interconnection networks.

We assume that each processor is assigned $\frac{n}{p}$ elements from the data set. The parallel algorithm also has two phases: the sample phase and the quantile phase. The number of runs per processor, r, equals $\frac{n}{pm}$. The sample phase of the parallel version is very similar to the sample phase of the sequential version. An additional step is required at the end for merging the local sample lists of all the p processors to form one global sorted sample list. The best algorithm for merging p lists depends on the underlying interconnection network of the parallel machine, the sizes of lists to be merged and the number of processors. We have investigated two algorithms which can be used to solve this problem: Bitonic merge and Sample merge. These are variations of the Bitonic sort [Bat68, KGGK94] and sample sort [LLS+93, KGGK94]. The only difference between Bitonic/sample sort and Bitonic/sample merge is that the initial sorting step is not required because the local lists are already sorted. The complexity of the Bitonic merge and the sample merge are given by $O(\delta(rs(1+\log p)\log p)+(1+\log p)\log p(\tau+\mu rs))$ and $O(\delta(s'+(p-1)\log rs+\beta rs\log p))+(1+\log p)\log p(\tau+\mu s')+2(\tau p+\mu\beta rs)$, respectively [LLS+93, KGGK94]. β is defined as the bucket expansion factor which is bounded by $\frac{3}{2}$. s' is defined as the size of the sample size which is used by the sample merge.

By merging the p sample lists, we form a globally sorted sample list of size prs such that processor i will have $s_{rsi},\ldots,s_{rsi+rs-1}$ elements. The quantile phase in the parallel version of the algorithm is very similar to the corresponding one in the sequential algorithm. The only difference is in the number of total runs. In the sequential algorithm, the number of the total runs is r, whereas the number of the total runs in the parallel algorithm is rp. We can estimate the upper and lower bounds of φ-quantile by using formulas (2) and (4) of section 2 and substituting rp instead of r. Note that lemmas 1 through 3 also hold for the parallel algorithm.

The time requirement of the parallel algorithm is the sum of the time required for each of the following steps:

- Reading the $\frac{n}{p}$ elements from the disk locally.
- Finding the rs sample points locally.
- Merging the r sample lists locally.
- Merging the p sample lists globally.
- Estimating the value of the φ-quantile.

Reading the $\frac{n}{p}$ elements from the disk takes $O(\frac{n}{p})$ time. Finding the sample points takes $O(rm\log s)$ time, using algorithms given in [FR75]. Merging the r samples can be done in $O(rs\log r)$ time. As discussed earlier, merging the p sample lists can be achieved by either the Bitonic merge or the sample merge. We denote the complexity of merging the p sample lists globally by $T(p,x)$ where p is the number of processors and x is the size of the lists on each processor. Estimating the upper and the lower bounds of the value of the quantile takes constant time. Thus, the total complexity of the algorithm is $O(\frac{n}{p}+rm\log s+rs\log r+T(p,rs))$. As in the sequential version, the total complexity to find q quantiles is $O(\frac{n}{p}+rm\log s+rs\log r+T(p,rs)+q)$ which equals $O(\frac{n}{p}+\frac{n}{p}\log s+\frac{n}{mp}s\log\frac{n}{mp}+T(p,rs)+q)$, since $r=\frac{n}{pm}$.

In case $\frac{m \log s}{s} \geq \log \frac{n}{pm}$, the total complexity of the algorithm is $O(\frac{n}{p} \log s + T(p, rs))$. The total complexity of the algorithm for different merging algorithms is given in table 8. We expect the Bitonic merge to have better performance for small data sets and small number of processors. In other cases the sample merge should perform better.

Table 8: The time requirement of the parallel algorithm using different merging algorithms

Using	The Time requirement
Bitonic	$O(\delta(\frac{n}{p} \log s + rs(\mathrm{b} + \log p) \log p) + (1 + \log p) \log p(\tau + \mu rs))$
Sample	$O(\delta(\frac{n}{p} \log s + s' + (p-1) \log rs + \beta rs \log p) + (1 + \log p) \log p(\tau + \mu s') + 2(\tau p + \mu \beta rs))$

3.1 Scalability Analysis

A detailed scalability analysis is done in a longer version of this paper [ARS97]. Using a formally defined scalability metric called *isoefficiency*, we have shown that OPAQ is scalable. The analysis shows that the sample merge version of the parallel algorithm is far more scalable than the bitonic merge version.

3.2 Experimental Results

We implemented the OPAQ algorithm on the IBM SP-2. Each node of the SP-2 is a RISC System/6000 module 390 with 128 MBytes of main memory. Each node is running AIX version 4. We experimented with the parallel version of the algorithm on data sets with uniform distribution only. The experimental results of the sequential version of the algorithm demonstrated that the accuracy of the algorithm does not significantly depend on the distribution of the data sets.

Figure 3 shows the execution time of the global merge phase using Bitonic merge and sample merge. The data sizes used are $1K$, $2K$, $4K$, $8K$, $16K$, $32K$, $64K$ and $128K$ per processor. The Bitonic merge outperforms the sample merge for small number of processors and small data sets. For large number of processors and large data sets, the sample merge outperforms the Bitonic merge. We only present results using sample merge for the rest of this section.

The number of elements per processor was varied from $0.5M$, $1M$, $2M$, and $4M$ to study the effect of scaleup, sizeup and speedup properties of our algorithm. This data was stored in the disks attached with the processors. The number of processors used were 1, 2, 4, 8 and 16. The sample size s and the run size m are set to $1K$ and $128K$ elements respectively (independent of the number of processors).

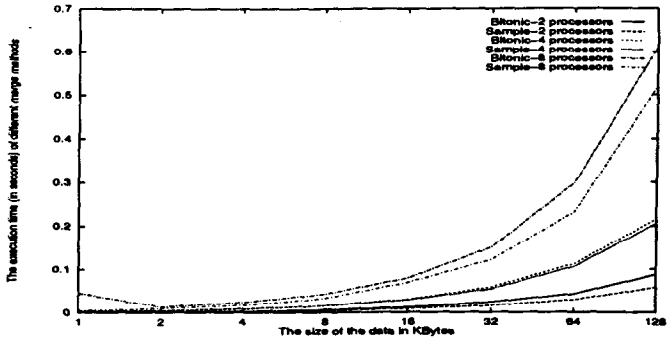

Figure 3: The execution time of different merge methods

Table 9: The RER_A produced by the parallel algorithm for different data sets

Dectile	Uniform Distribution						
	0.5M	1M	2M	4M	8M	16M	32M
10%	0.09	0.08	0.09	0.10	0.09	0.10	0.09
20%	0.09	0.08	0.09	0.10	0.09	0.09	0.09
30%	0.08	0.09	0.10	0.09	0.10	0.10	0.09
40%	0.10	0.10	0.09	0.09	0.09	0.09	0.09
50%	0.09	0.08	0.08	0.09	0.09	0.09	0.09
60%	0.09	0.09	0.10	0.09	0.09	0.09	0.09
70%	0.09	0.08	0.07	0.10	0.09	0.09	0.09
80%	0.09	0.09	0.09	0.10	0.09	0.09	0.09
90%	0.08	0.09	0.09	0.10	0.09	0.09	0.09

We conducted several experiments to determine the error rates produced by the algorithm for finding dectiles in different size data sets. Tables 9 and 10 show results (the reported data sizes are the total sizes of the data) for 8 processors. Our experimental results showed that the error rates produced were independent of data set size.

The algorithm spends around 50% of the total execution time in performing I/O. Table 11 shows the percentage of the I/O time to the total execution time for different data sizes and different machine sizes. Table 12 shows the fraction of the execution time of the different phases of the algorithm. The number of elements per processor is set to $4M$. The I/O time and sampling time take more than 83% of the total execution time of the algorithm and are relatively independent of the number of processors used. Hence, the algorithm should scale well for larger number of processors.

We did not invest any effort in optimizing the overlap in I/O and computation time. One can potentially reduce the overall time by overlapping part of the computation time with the I/O time.

Figure 4 shows that our algorithm is scalable. This is because the extra overhead of the parallel algorithm is the cost of the global merge. This cost is small com-

Table 10: The RER_L AND RER_N produced by the parallel algorithm for different data sets

Dectile	Uniform Distribution						
	0.5M	1M	2M	4M	8M	16M	32M
RER_L	0.62	0.62	0.54	0.61	0.53	0.54	0.51
RER_N	0.67	0.60	0.59	0.61	0.56	0.54	0.52

Table 11: The percentage of the I/O time to the total time for various number of elements per processor and various number of processors

Size	1 Proc.	2 Proc.	4 Proc.	8 Proc.	16 Proc.
0.5M	0.54	0.53	0.52	0.52	0.50
1M	0.53	0.40	0.52	0.51	0.50
2M	0.53	0.57	0.51	0.51	0.53
4M	0.52	0.49	0.51	0.52	0.51

pared to the cost of the other phases of the algorithm

Figure 5 shows that our algorithm has good sizeup characteristics. This is again due to the low cost of the global merge.

Our algorithm has a high speedup performance. This is also due to the low cost of the global merge. Figure 6 shows the speedup of our algorithm for a total of 4M elements.

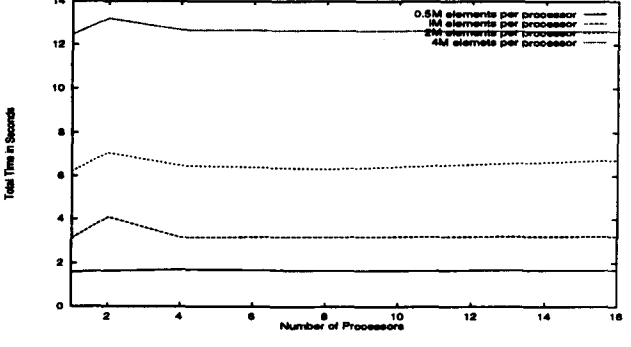

Figure 4: Scale-up of OPAQ

4 Conclusions

In this paper, we have presented and analyzed OPAQ, a new algorithm for estimating the φ-quantile value on sequential and parallel machines. OPAQ has the following characteristics:

- It requires only one pass over the data.
- It is deterministic.
- It produces good lower and upper bounds of the true value of the φ-quantile.
- It requires no a priori knowledge of the data set.

Table 12: The percentage of the execution time of the various phases of the algorithm to the total time for $4M$ elements per processor and various number of processors

Phase	1 Proc.	2 Proc.	4 Proc.	8 Proc.	16 Proc.
I/O	0.52	0.49	0.51	0.52	0.51
Sampling	0.47	0.44	0.47	0.46	0.45
Local Merge	0.004	0.051	0.003	0.004	0.009
Global Merge	0	0.002	0.005	0.010	0.015

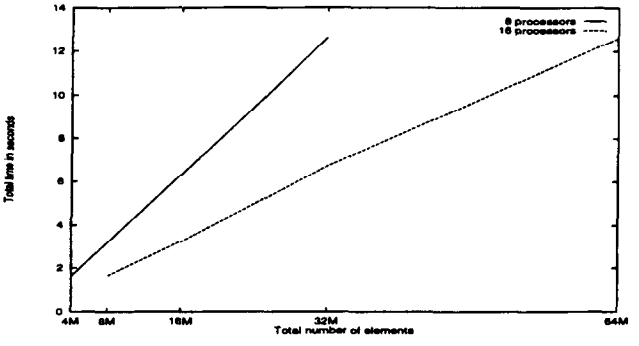

Figure 5: Size-up of OPAQ

- It has a scalable parallel formulation.
- The additional cost for each additional quantile beyond the first one is constant per quantile.

The computation time of our algorithm is linear in the size of the data set for a fixed number of quantiles and a given error rate. Further, it provides the flexibility of improving the accuracy of the results obtained by increasing the computational time.

The sorted sample list can obviously be used to estimate the rank of any arbitrary element in the whole data set. This does not require any extra passes over the entire data set.

It is easy to use the OPAQ algorithm to deal with new data incrementally. If the sorted samples are kept from the runs of the old data, one need only compute the sorted samples from the new runs and merge with the old sorted samples.

The OPAQ algorithm can be extended to find the exact value of a given quantile. This will require one extra pass over the data set. In the extra pass, we keep the elements which are in the interval $[e_\alpha^l..e_\alpha^u]$. We also count the number of elements which are less than e_α^l to find the rank of e_α^l, $R_{e_\alpha^l}$. The number of elements in the interval $[e_\alpha^l..e_\alpha^u]$ is less than or equal to $2\frac{n}{s}$ (by lemma 3). We can find the exact value of the quantile by sorting those elements. The exact value of the quantile is the element (in the sorted list) with rank $\alpha - R_{e_\alpha^l}$.

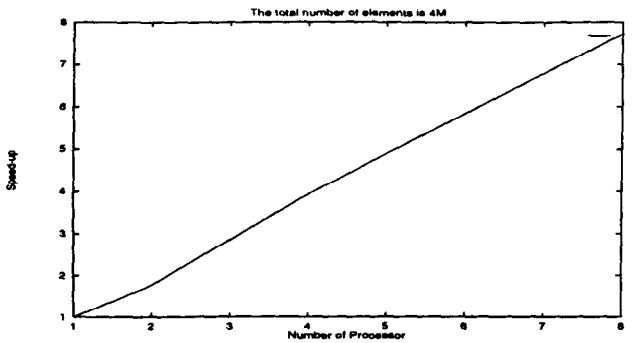

Figure 6: Speedup of OPAQ

In future work, we will overlap the I/O with computation. Since a large fraction of the total execution time is spent in I/O, we can significantly reduce the total execution time by overlapping the I/O and the computation. Moreover, we intend to investigate several important applications of quantiles using the OPAQ algorithm: database query optimizers, data mining (association rules and multi-dimensional similarity search [AS95, AIS93, AS96, ALSS95]), external sorting, and load balancing on multiprocessors.

References

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining Associations between Sets of Items in Massive Databases. *Proc. of the ACM SIGMOD Int'l Conference on Management of Data, Washington D.C*, pages 207–216, May 1993.

[ALSS95] R. Agrawal, K. Lin, H. S. Sawhney, and K. Shim. Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases. *Proc. of VLDB, Zurich*, 1995.

[ARS97] K. Alsabti, S. Ranka, and V. Singh. A One-Pass Parallel Algorithm for Accurately Estimating Quantiles for Disk-Resident Data. *http://www.cise.ufl.edu/~ranka/*, 1997.

[AS95] R. Agrawal and A. Swami. A One-Pass Space-Efficient Algorithm for Finding Quantiles. *Proc. of the 7th Int'l Conf. Management of Data (COMAD-95)*, Pune, India, 1995.

[AS96] R. Agrawal and R. Srikant. Mining Quantitative Association Rules in Large Relational Tables. *In Proc. of the ACM SIGMOD Conference on Management of Data, Montreal, Canada*, June 1996.

[B+72] M. Blum et al. Time Bounds for Selection. *Journal of Computers and Systems*, 7:4:448–461, 1972.

[Bat68] K. Batcher. Sorting Networks and their applications. *In Proceedings of the AFIPS Spring Joint Computing Conference*, Vol. 32, 1968.

[Coc77] W. G. Cochran. *Sampling Techniques.* John Wiley and Sons, New York, N.Y. 3rd edition, 1977.

[D+91] D. J. DeWitt et al. Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting . *1st Int'l Conf. on Parallel and Distributed Information Systems*, Miami Beach:280–291, 1991.

[FR75] R. W. Floyd and R. I. Rivest. Expected Time Bounds for Selection. *Communications of the ACM*, 18(3):165–172, 1975.

[GS90] A. P. Gurajada and J. Srivastava. Equidepth Partitioning of a Data Set based on Finding its Medians. *Technical Report TR-90-24, Computer Science Dept., Univ. of Minnesota*, 1990.

[JC85] R. Jain and I. Chlamtac. The P^2 Algorithm for Dynamic Calculation for Quantiles and Histograms Without Storing Observations. *CACM*, Vol. 28, No. 10:1076–1085, Oct. 1985.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc, 1994.

[Koo80] R. P. Kooi. The Optimization of Queries in Relational Databases. *PhD Thesis, Case Western Reserver University*, Sept. 1980.

[LLS+93] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):543–550, October 1993.

[MD88] M. Muralikrishna and D. J. Dewitt. Equi-Depth Histograms for Estimating Selectivity Factors for Multidimensional queries. *Proc. of ACM SIGMOD*, Chicago, Illinois:28–36, June 1988.

[MP80] J. I. Munro and M. S. Paterson. Selection and Sorting with Limited Storage. *Theoretical Computer Science*, 12:315–323, 1980.

[PIHS96] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved Histogram for Selectivity Estimation of Range Predicates. *Proc. of the 1996 ACM SIGMOD*, Montreal:294–305, June 1996.

[PS84] G. Piatetsky-Shapiro. Accurate Estimation of the Number of Tuples Satisfying a Condition. *ACM SIGMOD 84*, Boston:256–276, June 1984.

[SD77] B. W. Schmeiser and S. J. Deutsch. Quantile Estimation from Grouped Data: The Cell MidPoint. *Communications in Statistics: Simulation and Computation*, B6(3):221–234, 1977.

[Zip49] G.K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Reading, MA, 1949.

A The Properties of the Sample Points

In this appendix we give a more detailed explanation of the properties used in determining $e_\alpha{}^l$ and $e_\alpha{}^u$.

The first property is shown in figure 7. As a result of using regular sampling in deriving the sample points, it can be easily shown that each sample point s_i represents a sub-run of size $\frac{m}{s}$ elements. These elements are less than or equal to s_i.

Figure 8 shows the second property. Given the first property and the rs sample points are sorted, we conclude that there are at least $\frac{m}{s}$, $\frac{2m}{s}$,..., $\frac{im}{s}$ elements less than or equal to the sample points $s_1, s_2, \ldots, s_i$, respectively.

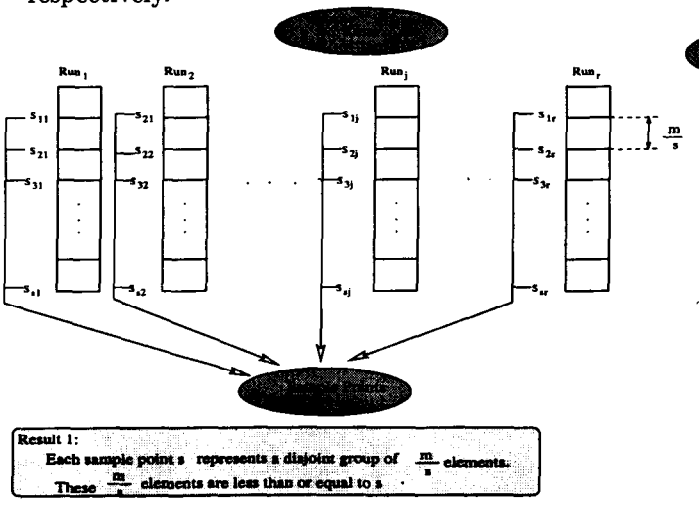

Figure 7: The derivation of the sample points from each run

There are at least $\frac{im}{s}$ elements less than or equal to the sample points s_i. In addition to that, there are at most $(r-1)(\frac{m}{s}-1)$ elements less than s_i. Each group of $\frac{m}{s}-1$ elements has a unique corresponding run other than s_i's. This property is shown in figure 9.

Thus the maximum number of elements less than s_i is given by $i\frac{m}{s}+(r-1)(\frac{m}{s}-1)$.

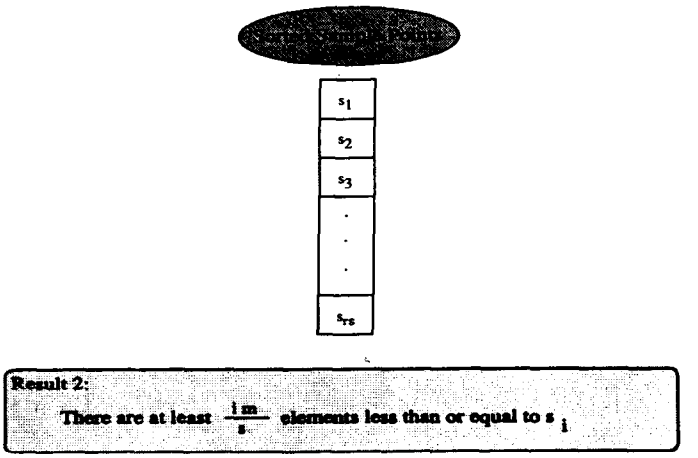

Figure 8: The minimum number of elements less than or equal to s_i

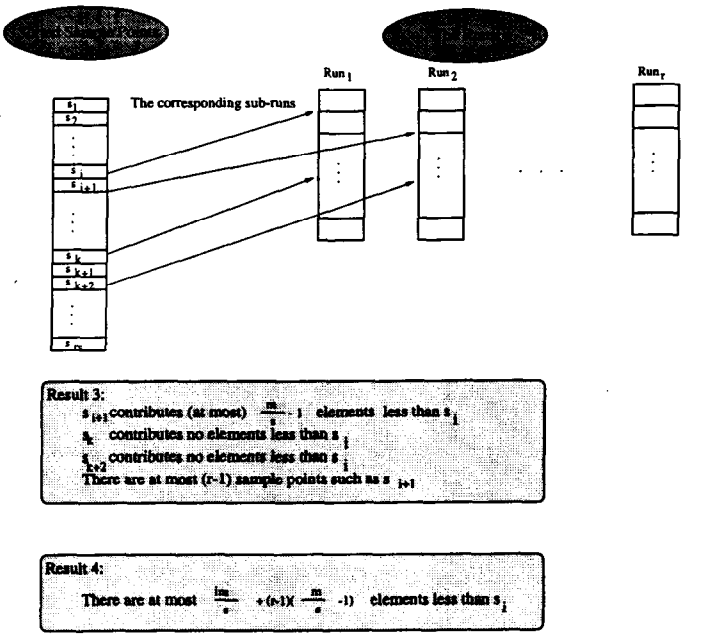

Figure 9: The maximum number of elements less than s_i