Percentile Finding Algorithm for Multiple Sorted Runs

Balakrishna R. Iyer
Database Technology Institute
IBM Corporation
San Jose, CA 95161-9023

Gary R. Ricard
Applications Business Systems
IBM Corporation
Rochester, MN 55901

Peter J. Varman
Elec. & Comp. Eng. Department
Rice University
Houston, TX 77251-1892

Abstract External sorting is frequently used by relational database systems for building indexes on tables, ordered retrieval, duplicate elimination, joins, sub-queries, grouping, and aggregation; it would be quite beneficial to parallelize this function. Previous parallel external sorting algorithms found in the database literature used a sequential merge as the final stage of the parallel sort. This reduces the speedup gained through parallelism in earlier stages of sort. The solution is to merge in parallel as well. Load balanced parallel two way merges and approximately load balanced parallel multi way merges are known. Measurements reported on parallel sorting that employs one of the approximate partitioning methods indicate that even if the sort keys are randomly distributed the load imbalance due to the approximation degrades speedup due to parallelism. Sort key value skews, known to occur in database workloads, can only exacerbate this problem. We give, prove and analyze an efficient exact method which can find any percentile of an arbitrary number of sorted runs. Application of our algorithm ensures load balance during the parallel merge. By removing the effect of skews of sort key values which caused loss of speed up in previous approaches our method can improve the speedup for parallel sorting on multiple processors. While we target our work to a parallel computer architecture of shared memory MIMD parallel processors, our results are also likely to be useful for other parallel computer architectures.

1 Introduction The need for database MIPS per database installation is outstripping the uniprocessor MIPS supplied by computer vendors. External sorting is frequently invoked by relational database systems for building indexes on tables, ordered retrieval, duplicate elimination, joins, sub-queries, grouping, and aggregation and is known to be a time consuming operation. External sorting on multiple processors is, therefore, an important and beneficial problem to be solved for relational database systems. For purposes of exposition and analysis we will assume a shared memory shared data computer architecture. Yet, the reader may find much of our work equally applicable to loosely coupled architectures.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

1.1 Background We start by examining some of the parallel sort algorithms in the database literature. Perhaps the simplest technique is to partition the sort key range prior to accessing the database. Selected rows from the database are assigned to different buckets, each bucket corresponding to a key range. Rows in different buckets are sorted in parallel and then the result concatenated or used for further processing. This simple technique is not practical for the following two reasons: (a) tight upper and lower bounds for the sort key range are not easily determined before the rows from a database are selected, and (b) it is difficult to partition the sort key range into partitions so that about the same number of rows will have sort keys belonging to any one partition, load balancing is compromised.

Bitton et. al. (BIT83) propose two external parallel sort algorithms for use in database systems that they call parallel binary merge and block bitonic sort. Both algorithms employ the binary two way merge of sorted runs. In a typical external sort for database systems, sorted runs are first created as temporary relations on disk and then merged. If we use algorithms that rely on a two way merge the number of I/O's, temporary relation inserts and fetches per sorted row will be of the order of $\log_2(NR)$ where NR is the number of initial sorted runs created. Both I/O's and number of temporary relation interactions are expensive and they make algorithms based on a two way merge expensive, suggesting that the two algorithms are primarily useful for internal sort where there is no I/O or temporary relation interaction. Even for parallel internal sorting, Bitton's parallel binary merge algorithm suffers from the drawback that its last phase is essentially a sequential binary merge and it is therefore impossible to finish the sort earlier than the time for one sequential pass through the entire data regardless of the number of processors. While it may not be possible to avoid examining all the data for sorting, we must avoid examining it all serially if we are to obtain speedup linear in the number of processors (over a reasonable range of values for the number of processors). Valduriez and Gardarin (VAL84) generalized the parallel binary merge algorithm to a k-way merge algorithm. In their algorithm p processors will merge different sets of runs in parallel using a k-way merge. We then have p runs. These are merged (assuming $p < k$) sequentially on a single processor. A pipelined version of this algorithm has been proto-

typed (BEC88) and another implemented in a product (GRA88). The bottleneck due to a sequential merge is still present in all these algorithms.

The idea of employing multiple processors to merge the same runs appears in the literature in the context of merging two runs on two processors. Based on the pioneering work of Batcher (BAT68), an efficient algorithm was proposed (AKL87) to break each of two runs into two partitions so that key values in the first partition of each run were no greater than key values in the second partition of either run (magnitude requirement). Also, the sum of the number of elements in the first partition of the two runs was half the sum of the total number of elements in both runs (size requirement). The first partition of each run was merged on one processor and the second partition of each run is merged independently on another processor. Although the method meets the requirements for load balancing a parallel sort if there are two runs and two processors, it provides no obvious or intuitive algorithmic method for extension to an arbitrary number of equal sized partitions for an arbitrary number of runs. The idea of employing an arbitrary number of processors to merge two runs appears only recently (VAR88) (FRA88) (DEO88). A method to partition two runs into an arbitrary number of equal parts corresponding to percentile ranges is known. A multiprocessor multi-way merge can be implemented by a sequence of multiprocessor two-way merges. While this method gets around the bottleneck due to a final sequential merge, the I/O overheads incurred are proportional to $\log_2(NR)$, where NR is the number of sorted runs. The idea of employing an arbitrary number of processors to merge as many runs as there are processors (in one pass) appears to be first due to Quinn (QUI88). He proposed that m processors be used to merge m runs in parallel, each processor employing an m-way merge on $1/m$'th of the data.

1.2 Problem Statement How we obtain the m-way partitioning of an **arbitrary number** of runs for arbitrary m is the subject of this paper. We show how to solve the problem of partitioning an arbitrary number of m sorted runs into two parts such that one partition has an arbitrary fraction f of the total number of elements. Elements of one partition have keys greater than any element of the other partition. Recall that previous known exact algorithms can partition only **two runs** into multiple equal parts. Initially, we simplify the problem by making all runs equal in size, restricting the size to a power of two and pad the runs with $+\infty$ and $-\infty$ on either end (all other elements in the run must be strictly greater than $-\infty$ and strictly less than $+\infty$). The values taken by the fraction f are also restricted. An algorithm is given to find the partition of m such runs containing fraction $f = i/m$ of the data, where $1 \leq i \leq m$, and where the initialization (basis) step involves examining up to $O(m^2)$ elements.

1.3 Previous Solutions First we discuss previous solutions. One simple method to solve the partitioning problem approximately is to use a sampling technique (LOR89). This involves sampling the keys in the m runs, sequentially merging the samples, finding an exact partition of the samples (through a search of the merged runs of samples), and determining the exact m key ranges that partition the samples. The m keys that define the ranges are then used as indices into the runs to define the m partitions. Sampling techniques are discussed in (VIT85) and (OLK86).

An approximate technique that partitions the runs after they are formed has been proposed by Quinn (QUI88). Equally spaced apart keys from the first run are used to index into and partition the remaining runs. We illustrate this technique by means of an example of 3 runs and 3 partitions in Figure 1. The keys 9 and 15 from run 1 are selected since they partition it into three sequences of equal length. The two keys are used to partition the remaining two runs as shown in Figure 1. The first partitions of each run totalling to 11 keys will be merged on processor 1. The second partitions of each run totalling to 8 keys will be merged on processor 2, and the third partitions of each totalling to 8 keys will be merged on processor 3. This method does not exhibit exact load balancing. Experiments by Quinn show that the speed up of parallel sorting by employing this algorithm levels off beyond 10 processors for sorting a random permutation of 10,000 keys. Quinn attributes the loss of speed-up to the approximation and resulting loss of load balancing during the merge. Skewed distribution of sort key values in rows selected from a database are not uncommon in practice. In these cases the risk of the approximation is even greater. Quinn cites an example where the first run contains elements that are all smaller than the elements in all the other runs. Approximate partitioning leads to a merge time that is comparable to a sequential merge. Speedup would be severely compromised in other similar situations.

In order to ensure perfect load balance and prevent loss of speed up during the final parallel merge, an alternate and exact technique is proposed to partition the runs after they are formed. We give an algorithm that divides the m arbitrary sized sorted runs into two partitions such that one partition contains any given arbitrary fraction f of the total number of elements in the runs, and all elements in that partition are no greater than the elements in the other. By employing this algorithm repeatedly for $f = \frac{1}{m}, \frac{2}{m}, \ldots, \frac{(m-1)}{m}$

the m-way partitioning problem can be solved. It will be shown that as number of rows grows, the complexity of this algorithm (for a small number of processors) grows only logarithmically as the number of rows in each of the m runs. This is why we ignore the cost of partitioning the runs when estimating the elapsed time of the parallel sort algorithm. Recently, we became aware of an algorithm with asymptotically optimal number of comparisons for this problem (FRE82). The algorithm is quite complex and has a slower rate of convergence than the one proposed here, and requires significantly greater (about six times) the number of I/Os.

In the execution of complex queries in relational database systems we can visualize multiple processors employed in various ways. Specifically, for queries requiring ordering, e.g. sort/merge joins, multiple processors may be used for selecting rows and sorting different partitions of a relation or a composite produced as a result of previous steps (IYE88). As illustrated in Figure 2, the access of data from different partitions and the formation of runs is easily parallelized. Different partitions may be assigned to different processors. The assigned processor selects, sorts and merges rows belonging to a partition independent of other partitions. At the end, we need to merge (or merge/join) in parallel with multiple input runs. Our way to parallelize the merge (or merge/join) operation is to divide the sorted runs into equal parts based on key ranges and assign one part to each processor. Each processor merges (or joins) its partition of rows, roughly corresponding to different key ranges. (For joins it is possible to extend our problem and solution to ensure that there is no overlap among the key ranges. For sorting the smallest key in a range may appear as the largest key in the adjacent range.)

In the next section we state the algorithm for a restricted case of the problem, give its proof and an example. The generalized algorithm is discussed in section 3. We analyze the complexity of the algorithm in section 4. Concluding remarks are found in section 5.

Equal Sized Lists Algorithm

2.1 Problem The casual reader may skip the following subsections to the subsection containing the example. We start with a formal definition of the simplified problem. Let
$A_r = a_{r,0}, a_{r,1}, a_{r,2}, \ldots, a_{r,N}$, $1 \le r \le m$, $N = 2^k$, be m ascending sequences, with $a_{r,0} = -\infty$, $a_{r,N} = +\infty$. , and $+\infty > a_{r,i} > -\infty$ for $i > 0$. The problem is to find for each r, $1 \le r \le m$, an index i_r that defines a prefix $\hat{A}_r$ of A_r with the properties 1 and 2 below.
$\hat{A}_r = a_{r,0}, a_{r,1}, a_{r,2}, \ldots, a_{r,i_r}$ such that:

1. $\sum_{r=1}^{m} ||\hat{A}_r|| = f \sum_{r=1}^{m} ||A_r|| = fm(N+1)$ for $f = \frac{i}{m}$, $1 \le i \le m$ (formal statement of the partition size requirement in terms of the cardinalities $||.||$) and
2. $\forall x, x \in \hat{A}_r$, $1 \le r \le m$, and $\forall y, y \in A_t, y \notin \hat{A}_t$, $1 \le t \le m$, $x \le y$. (formal statement of the magnitude requirement).

We refer to $I = \{i_r, 1 \le r \le m\}$ as the partitioning function of $\{A_r, 1 \le r \le m\}$ and to $\{\hat{A}_r \mid 1 \le r \le m\}$ as the **leading partition** of $\{A_r \mid 1 \le r \le m\}$. The m sequences defined should be interpreted as the sequence of sort keys in the m runs. Element $a_{r,i}$ of the sequence is the sort key of the i 'th row, of the r'th sorted run. The partitioning function is a set of indices into the m sequences that define where sequences must be split. The index i_r is the position of the sort key with the largest value in A_r and which belongs to the partition containing the smallest f fraction of the entire data. We refer to such an element of the sequence as a **boundary element** (for fixed length rows from a database sequentially stored on a disk or tape, it is straightforward to calculate the disk or tape address from the the index or the boundary element in the sequence of sort keys, for unequal sized rows we will need to use some form of indexing into the run).

Let A_r^p denote the subsequence of A_r consisting of $(2^p + 1)$ equally spaced apart elements, i.e.,

$$A_r^p = a_{r,0}, a_{r,N(p)}, a_{r,2N(p)}, \ldots, a_{r,N},$$

where $N(p) = N/2^p$. Let $I^p = \{i_1^p, i_2^p, \ldots, i_m^p\}$ denote the partitioning function of $\{A_r^p, 1 \le r \le m\}$. Recalling that $N = 2^k$, we note that $A_r^k = A_r$ and I^k is the partitioning function that we are looking for. We solve the problem by successively computing the partitioning functions I^{p-1} from I^p using $O(m)$ I/O's and $O(m \log_2 m)$ comparisons.

2.2 Definitions Consider the subsequence A_r^p and its leading partition $\hat{A}_r^p$, $1 \le r \le m$. These consist of the elements
$A_r^p = a_{r,0}, a_{r,N(p)}, a_{r,2N(p)}, \ldots, a_{r,N}$
$\hat{A}_r^p = a_{r,0}, a_{r,N(p)}, a_{r,2N(p)}, \ldots, a_{r,i_r^p}$,
where $I^p = \{i_1^p, \ldots, i_m^p\}$ is the partitioning function of $\{A_r^p, 1 \le r \le m\}$.

For the set $\{A_r^p, 1 \le r \le m\}$ and partitioning function I^p, let a_{max}^p denote the largest of the m boundary elements, i.e., $a_{max}^p = \max_{1 \le r \le m}(a_{r,i_r^p})$. A consequence of the simplifications made in this subsection is that the leading partition of $\{A_r^p, 1 \le r \le m\}$ will always

contain at least one element from each subsequence A_r^p. That is:

$$i_r^p \geq 0, \text{ for } 1 \leq r \leq m. \tag{1}$$

This follows because of the restrictions we placed on the values that can be taken by the fractions f and the padding with $-\infty$'s and by the appropriate selection of the basis step to be described later. None of these assumptions will be needed in the generalized algorithm in the next subsection which embodies the same key idea as the algorithm we give next to solve the restricted problem.

At the start of the current iteration, we are given the partitioning function I^p of $\{A_r^p, 1 \leq r \leq m\}$ and a_{max}^p. The former satisfies both the partition size and magnitude requirements. That is:

$$\sum_{r=1}^{m} ||\hat{A}_r^p|| = f \sum_{r=1}^{m} ||A_r^p|| \text{ (partition-size rqrmnt.) and}$$

$$a_{r,j} \leq a_{u,k} \text{ whenever} \tag{2}$$

$$a_{r,j} \in A_r^p,\ 1 \leq r \leq m,\ 0 \leq j \leq i_r^p,$$
$$a_{u,k} \in A_u^p,\ 1 \leq u \leq m,\ i_u^p < k \leq N \text{ (magnitude rqrmnt.)}$$

The goal of the current iteration is to find from this partition of $\{A_r^p, 1 \leq r \leq m\}$ the partitioning for $\{A_r^{p+1}, 1 \leq r \leq m\}$ satisfying both partition size and magnitude requirements.

2.3 Algorithm and Proof First, note the cardinalities of the two subsequences A_r^p and A_r^{p+1} are related as $\sum_{r=1}^{m} ||A_r^{p+1}|| = \left(2\sum_{r=1}^{m} ||A_r^p||\right) - m$ for $p \geq 1$, since A_r^{p+1} is obtained by inserting an element between every consecutive pair of elements of A_r^p and from (1).
Let $X_r^{p+1} = a_{r,0}, a_{r,N(p+1)}, a_{r,2N(p+1)}, \ldots, a_{r,i_r^p}$ be the sequence obtained by taking the leading partition of A_r^p (i.e., $\hat{A}_r^p$) and adding to it the elements halfway between every two consecutive elements (note $N(p+1) = N(p)/2$). All the elements in X_r are equally spaced apart by $N(p+1)$. Its cardinality is easily computed as follows:
$\sum_{r=1}^{m} ||X_r^{p+1}|| = \left(2\sum_{r=1}^{m} ||\hat{A}_r^p||\right) - m = \left(2f \sum_{r=1}^{m} ||A_r^p||\right) - m$
$= f\left(\sum_{r=1}^{m} ||A_r^{p+1}|| + m\right) - m = \left(f \sum_{r=1}^{m} ||A_r^{p+1}||\right) - m(1-f).$
At least, in cardinality, we are close to the partitioning we want. We are only off by $m(1-f)$ elements.

For each r, $1 \leq r \leq m$, consider the new element of A_r^{p+1} that is $N(p+1)$ away from (greater than) the boundary element of A_r^p, i.e., the new element $a_{r,i_r^p+N(p+1)}$. Some of these could be smaller than a_{max}^p, the largest boundary element of the previous iteration, and may have to be included in the leading partition of $\{A_r^{p+1}, 1 \leq r \leq m\}$ to satisfy the magnitude requirement. For each r, $1 \leq r \leq m$, for which this is true (i.e., $a_{r,i_r^p+N(p+1)} \leq a_{max}^p$), we add the elements to the sequence X_r^{p+1} to obtain the sequence Y_r^{p+1}. If the condition is false, we let $Y_r^{p+1} = X_r^{p+1}$. Let the total number of elements so added be s. If we let j_r^{p+1} denote the index of the largest element in Y_r^{p+1} , then in the first case, $j_r^{p+1} = i_r^p + N(p+1)$, otherwise $j_r^{p+1} = i_r^p$. Its value will be further refined to give i_r^{p+1} next.

Let us compute the cardinality of the expanded sequence:

$$\sum_{r=1}^{m} ||Y_r^{p+1}|| = \left(\sum_{r=1}^{m} ||X_r^{p+1}||\right) + s \tag{3}$$

$$= \left(f\sum_{r=1}^{m} ||A_r^{p+1}||\right) - m(1-f) + s$$

It should be clear from the construction that every element of Y_r^{p+1} is no greater than a_{max}^p , and that every element of A_r^{p+1} not included in Y_r^{p+1} is at least as large as a_{max}^p . In other words Y_r^{p+1} is a leading partition of $\{A_r^{p+1}, 1 \leq r \leq m\}$ that satisfies the magnitude requirement. A formal proof follows:
Claim: If $a_{r,j} \in Y_r^{p+1}$ then $a_{r,j} \leq a_{max}^p$
Proof: If $Y_r^{p+1} \neq X_r^{p+1}$ then $a_{r,j} \leq a_{r,i_r^p+N(p+1)} \leq a_{max}^p$
If $Y_r^{p+1} = X_r^{p+1}$ then $a_{r,j} \leq a_{r,i_r^p} \leq a_{max}^p$.

Claim: If $a_{r,j} \in A_r^{p+1}$ and $a_{r,j} \notin Y_r^{p+1}$ then $a_{r,j} \geq a_{max}^p$.
Proof: If $Y_r^{p+1} \neq X_r^{p+1}$ then
$a_{r,j} \geq a_{r,i_r^p+N(p+1)+N(p+1)} = a_{r,i_r^p+N(p)}$. From (2)
$a_{r,i_r^p+N(p)} \geq a_{max}^p$ thus $a_{r,j} \geq a_{max}^p$. If $Y_r^{p+1} = X_r^{p+1}$ then
$a_{r,j} \geq a_{r,i_r^p+N(p+1)} > a_{max}^p$ by definition of Y_r^{p+1}.

Thus $\{Y_r^{p+1}, 1 \leq r \leq m\}$ defines a leading partition of $\{A_r^{p+1}, 1 \leq r \leq m\}$ which satisfies the magnitude requirement. The identification of $\{Y_r^{p+1}, 1 \leq r \leq p\}$ takes O(m) I/O's and O(m) comparisons.

Next we check if the partition-size requirement is satisfied; if not we adjust the number of elements to satisfy the size requirement.

Case 1: $s = m(1-f)$
From (3) it follows that $\sum_{r=1}^{m} ||Y_r^{p+1}|| = f\sum_{r=1}^{m} ||A_r^{p+1}||$, and hence $\hat{A}_r^{p+1} = Y_r^{p+1}$ and the partition-size requirement is satisfied. In this case: $i_r^{p+1} = j_r^{p+1}$, $1 \leq r \leq m$.

Case 2: $s > m(1-f)$
From (3), we know $\{Y_r^{p+1}, 1 \leq r \leq m\}$ is close to being the correct leading partition of $\{A_r^{p+1}, 1 \leq r \leq m\}$

except that it has $s - m(1 - f)$ too many elements. While preserving the magnitude requirement, we can satisfy the partition-size requirement for $\{A_r^{p-1}, 1 \le r \le m\}$ if we remove the $s - m(1 - f)$ largest[1] elements from $\{Y_r^{p-1}, 1 \le r \le m\}$. The elements remaining in $\{Y_r^{p-1}, 1 \le r \le m\}$ after the largest $s - m(1 - f)$ elements are discarded is a leading partition of $\{A_r^{p-1}, 1 \le r \le m\}$ that satisfies both requirements. For every element removed from Y_r^{p-1}, j_r^{p-1} is decremented by $N(p + 1)$. Hence, if α_r elements are removed from Y_r^{p+1}, then $i_r^{p-1} = j_r^{p-1} - \alpha_r N(p + 1)$, $1 \le r \le m$. Since, $0 \le s < m$ and $0 < f < 1$, the above computation can be bounded by O(m) I/O's and O($m \log m$) comparisons. The $s - m(1 - f) + 1$ 'th largest element in $\{Y_r^{p-1}, 1 \le r \le m\}$ is also found and set to a_{max}^{p+1}, for use in the next iteration. Note: $\sum_{r=1}^{m} ||\hat{A}_r^{p-1}|| = \left(\sum_{r=1}^{m} ||Y_r^{p-1}||\right) - (s - m(1 - f))$ $= f \sum_{r=1}^{m} ||A_r^{p-1}||$ from (3), as needed for the partition size requirement. Also, since the largest $s - m(1 - f)$ elements are moved, the magnitude requirement remains satisfied.

Case 3: $s < m(1 - f)$
From (3), we know $\{Y_r^{p-1}, 1 \le r \le m\}$ is close to being the correct leading partition of $\{A_r^{p+1}, 1 \le r \le m\}$ except that it contains $m(1 - f) - s$ too few elements. While preserving the magnitude requirement, we can satisfy the partition-size requirement for $\{A_r^{p-1}, 1 \le r \le m\}$ by moving the $m(1 - f) - s$ smallest[2] elements from $\{A_r^{p+1} - Y_r^{p+1}, 1 \le r \le m\}$ into $\{Y_r^{p+1}, 1 \le r \le m\}$. For every element moved into Y_r^{p+1}, j_r^{p+1} is incremented by $N(p + 1)$. Hence, if α_r elements are moved into Y_r^{p-1}, then $i_r^{p+1} = j_r^{p-1} + \alpha_r N(p + 1)$, $1 \le r \le m$. As above, the computation can be bounded by O(m) I/O's and O($m \log m$) comparisons. The largest of the $m(1 - f) - s$ elements moved is found and set to a_{max}^{p-1}, for use in the next iteration. Note: $\sum_{r=1}^{m} ||\hat{A}_r^{p+1}|| = \left(\sum_{r=1}^{m} ||Y_r^{p+1}||\right) + m(1 - f) - s$ $= f \sum_{r=1}^{m} ||\hat{A}_r^{p-1}||$ from (3), thus satisfying the partition size requirement. Since the smallest elements are moved into Y_r^{p+1}, the magnitude requirement remains satisfied.

If the induction is then applied, described as per the algorithm, then after $k = \log_2 N$ iterations, a leading partition of $\{A_r, 1 \le r \le m\}$ that satisfies both requirements is obtained.

2.4 Basis Step Let $p = \max\left(0, \text{ceil}\left(\log_2\left(\frac{1-f}{f}\right)\right)\right)$.
By brute force, examine all the elements in A_r^p, determine $\hat{A}_r^p$, and set the initial values of i_r^p. Note that in the basis step we are considering at least $\frac{m}{f}$ elements because $\sum_{r=1}^{m} ||A_r^p|| = \frac{mN}{(N/2^p)} + m = m2^p + m$ $\ge \frac{m}{f} - m + m = \frac{m}{f}$. There are exactly $m - \infty$'s in our m sequences equal in number to $f\frac{m}{f}$, the number of elements in the leading partition of $\{A_r^p, 1 \le r \le m\}$. Hence $i_r^p \ge 0$ as required by (1).

2.5 Example Consider as an example the four sequences given below, each consisiting of 9 elements. We would like to find a partition containing the 50'th percentile of the elements contained in the four sorted sequences.

```
     A1    A2    A3    A4

   |-inf  -inf  -inf  -inf |
   -------------------------
      x     x     x     x

      x     x     x     x

      x     x     x     x

      x     x     x     x

      x     x     x     x

      x     x     x     x

      x     x     x     x
   -------------------------
   |+inf  +inf  +inf  +inf |
```

In the basis step we examine only the first and last element of each sequence, and by brute force find the 50'th percentile of the 8 elements. For each sequence, each pair of elements considered are 8 elements apart. Because (in the simplified algorithm) we require the first element of each sequence to be $-\infty$ and the last to be $+\infty$ the boundary marking the 50'th percentile of the elements considered is as depicted above.

1 A tie between elements in the same sequence is broken by picking the element with the larger index. A tie between elements in different sequences is broken by picking the element from the larger indexed sequence.

2 A tie between elements in the same sequence is broken by picking the element with the smaller index. A tie between elements in different sequences is broken by picking the element from the smaller indexed sequence.

In the next iteration below, we consider elements 4 apart from each sequence. This introduces 4 new elements 27, 13, 5 and 10 from sequences A_1 , A_2, A_3 and A_4, respectively. The two largest among these 4 elements are assigned to the upper partition that comprises the 'smaller' 50'th percentile of elements considered and the remaining two elements are assigned to the lower partition or the 'larger' 50'th percentile as follows.

```
     A1    A2    A3    A4

 |-inf  -inf  -inf  -inf |
 -------------           |
     x     x |  x     x  |
             |           |
     x     x |  x     x  |
             |           |
     x     x |  x     x  |
 ------------|           |
 |  27    13 ||  5    10 |
 |           |------------
 |   x     x |  x     x
 |           |
 |   x     x |  x     x
 |           |
 |   x     x |  x     x
 |           -------------
 |+inf  +inf  +inf  +inf |
```

We keep track of the largest element belonging to the upper partition, 10. In the next iteration we consider elements 2 apart from each sequence. These elements as well as the partitioning from the previous iteration is given below.

```
     A1    A2    A3    A4

 |-inf  -inf  -inf  -inf |
 -------------           |
     x     x |  x     x  |
             |           |
    20     2 |  3     8  |
             |           |
     x     x |  x     x  |
 ------------|           |
 |  27    13 ||  5    10 |
 |           |------------
 |   x     x |  x     x
 |           |
 |  29    15 |  6    12
 |           |
 |   x     x |  x     x
 |           -------------
 |+inf  +inf  +inf  +inf |
```

Without examining, we conclude that the elements 3 and 8 of sequences A_3 and A_4 are no greater than 10 from the knowledge that A_3 and A_4 are sorted sequences. Similarly without examination, elements 29 and 15 of sequences A_1 and A_2 are known to be no less than 10. The only in-doubt elements are 20, 2, 6 and 12 belonging to the the four sequences. We examine each and pick those less than 10 to be included in the upper (smaller half) partition and the others to the lower partition. Since there are exactly two out of four in-doubt elements less than 10, we have the new partition as follows.

```
     A1    A2    A3    A4

 |-inf  -inf  -inf  -inf |
 -------                 |
     x |  x     x     x  |
 ------|                 |
 |  20 ||  2     3     8 |
 |      |-------         |
 |   x |  x  |  x     x  |
 |      -------|         |
 |  27    13 ||  5    10 |
 |           ||     ------
 |   x     x ||  x |  x
 |           ||    | -----
 |  29    15 ||  6 | |12 |
 |           |------ |   |
 |   x     x |  x    | x |
 |           ---------   |
 |+inf  +inf  +inf  +inf |
```

Note that of the elements considered so far we have found the 50'th percentile. The largest element in the upper partition continues to be 10. In the next step we consider all elements.

```
     A1    A2    A3    A4

 |-inf  -inf  -inf  -inf |
 -------                 |
     4 |  1     2     7  |
 ------|                 |
 |  20 ||  2     3     8 |
 |      |-------         |
 |  21 |  9  |  4     9  |
 |      -------|         |
 |  27    13 ||  5    10 |
 |           ||     ------
 |  28    14 ||  6 |  11
 |           ||    | -----
 |  29    15 ||  6 | |12 |
 |           |------ |   |
 |  31    19 |  7    |21 |
 |           ---------   |
 |+inf  +inf  +inf  +inf |
```

Again we can argue that there are only 4 in-doubt elements, viz., 4, 9, 7 and 11 from sequences A_1, A_2, A_3

and A_4, respectively. Three of these elements are less than 10 and are included in the upper partition and the other to the lower by using the method of the previous iteration. Such a partitioning has the deficiency that there are 19 elements in the upper partition and only 17 in the lower partition as given next.

```
         A1    A2    A3    A4

     |-inf  -inf  -inf  -inf |
     |                       |
     |  4     1     2     7  |
     -------                 |
       20 |   2     3     8  |
          |                  |
       21 |   9     4     9  |
          --------           |
       27    13  |  5    10  |
                 |   -------
       28    14  |  6 |  11
                 |    |
       29    15  |  6 |  12
                 |    |
       31    19  |  7 |  21
                 ------
     +inf  +inf  +inf  +inf
```

The imbalance is easily adjusted by picking the largest element from the upper partition, 10, and moving it to the lower. This solution of the 50'th percentile finding problem is as follows.

```
         A1    A2    A3    A4

     |-inf  -inf  -inf  -inf |
     |                       |
     |  4     1     2     7  |
     -------                 |
       20 |   2     3     8  |
          |                  |
       21 |   9     4     9  |
          --------    -------
       27    13  |  5 |  10
                 |    |
       28    14  |  6 |  11
                 |    |
       29    15  |  6 |  12
                 |    |
       31    19  |  7 |  21
                 ------
     +inf  +inf  +inf  +inf
```

3 Unequal Sized Lists Algorithm From a practical point of view, it is important to remove all the restrictions placed in the previous section. To accomplish this, the problem is restated and we derive a generalized algorithm from the simpler algorithm. The generalized algorithm handles arbitrary sized runs (not necessarily equal in size), arbitrary fraction of data f and does not assume infinity padding. The basis step is of order $O(m)$.

Let $A_r = a_{r,1}, a_{r,2}, a_{r,3}, \dots, a_{r,N_r}$, $1 \leq r \leq m$, $N_r = ||A_r||$, be m ascending sequences. Analogous to Section 2.1 the problem is to find i_r , $1 \leq r \leq m$ for $\hat{A}_r = a_{r,1}, a_{r,2}, a_{r,3}, \dots, a_{r,i_r}$ ($i_r = 0$ implies $\hat{A}_r$ is a null sequence.) such that:

1. $\left| \sum_{r=1}^{m} ||\hat{A}_r|| - f \sum_{r=1}^{m} ||A_r|| \right| \leq 1$, $0 \leq f \leq 1$ (partition-size requirement), and
2. $\forall x, x \in \hat{A}_r$, $1 \leq r \leq m$, $\forall y, y \in A_t$, $y \notin \hat{A}_t$, $1 \leq t \leq m$, $x \leq y$ (magnitude requirement).

Since we allow for any arbitrary fraction f, the partition-size requirement can be met up to a difference of 1. We allow being off in cardinality by (up to) 1 from the exact fraction. Let us denote $I = \{i_r, 1 \leq r \leq m\}$ as the partitioning function of $\{A_r, 1 \leq r \leq m\}$.

We built on the algorithm that we discussed earlier for the constrained problem on equal sized lists. We will avoid padding the sequences with $+\infty$ or $-\infty$ by selectively including more and more of the m sequences during the iterations. Before the end of the last iteration all of the sequences will be included. Generally, we postpone the inclusion of smaller sequences until later in the iteration. Specifically, the problem is solved by obtaining a series of solutions $I^p = \{i_r^p, 1 \leq r \leq m\}$, successively for $p = 0, 1, 2, \dots, k_{max} - 1$ given a basis step at $p = 0$. The number of iterations $k_{max} = \max_{1 \leq r \leq m}(k_r)$ where $k_r = \text{floor}(\log_2(N_r)) + 1$ and[3] I^p is the partitioning function of a selected subsequence of selected sequences. At the end of the p'th iteration all sequences that have at least $2^{k_{max}-p-1}$ elements would have been included. The refined partitioning function I^{p+1} is found from the partitioning function I^p in the p'th iteration using $O(m)$ I/O's and $O(m \log_2 m)$ comparisons. The required definitions, the algorithm and an example are given in (IYE89).

4 Time Complexity Analysis All logarithms in the following discussion are base 2. Let p denote the

[3] Floor($\log_2(x)$) where x is a positive integer can be calculated using only integer arithmetic

number of processors. To partition m sequences into two partitions the number of iterations required is $\log(N_{max})$, where N_{max} is the number of elements in the longest sequence since elements are added in power-of-2 increments each iteration and, by definition, the algorithm must iterate $\log(N_{max})$ times to determine a partition of all elements. Each iteration involves $O(m)$ I/Os and no more than $O(m \log m)$ comparisons. This can be seen as follows. Of all the elements that are under consideration for partitioning during an iteration, the element immediately following the current partition boundary in each sequence must be inspected to determine if it must be moved into the leading partition. This requires at most $m-1$ comparisons and $m-1$ I/Os. The $m \log m$ term arises for the number of comparisons in Case 2 and Case 3 where at most m largest or smallest elements must be determined such that the partition-size requirement is satisfied. Note, this involves only $O(m)$ I/Os. Thus the worst case order of the number of I/Os is $O(m \log N_{max})$ and the number of comparisons is $O(m \log m \log N_{max})$. In the best case, the number of I/Os and comparisons for an iteration are both $O(m)$ and occurs if the iteration of the algorithm executes Case 1, thereby eliminating the need for determining up to m smallest or largest elements in Case 2 and Case 3. For database applications the number of rows to be sorted is much larger than the number of processors, and the cost of partitioning is far outweighed by the cost of merging the runs as discussed next.

Once the partitioning is found, each of the p processors can independently merge the portions of the m lists in its partition. (The following analysis assumes that the I/Os can also be parallelized, i.e., the I/O subsystem can support p concurrent accesses to m lists.) This can be done in a single pass using an m way merge using $O((N \log m)/p)$ comparisons per processor and $O(N/p)$ I/Os per processor, where N is the total number of elements to be merged. It is often the case in database applications that the number of I/Os primarily determines the execution time of a query. A sequential algorithm would require $O(N \log m)$ comparisons and $O(N)$ I/O's for the merge. For cases of practical interest for database systems the number of processors $p << N$ the number of rows to be sorted, and the number of runs to be partitioned $m = p$, since the processors can sort independently without interference until p runs are obtained (VAL84). We assume $m = p$ in the following discussion. The cost of partitioning itself is negligible; what matters is the merge cost.

There have previously been two general types of approaches to solving this partitioning problem. The first is to partition pairs of sequences exactly. This leads to workload balance and minimal contention for each 2-way merge but requires $\log(p)$ merge passes to complete. While any merging algorithm (AKL87, FRA88, VAR88, DEO88) which requires parallel 2-way merges can be done with $O((N \log m)/p)$ comparisons per processor, the number of I/Os per processor is $O((N \log p)/p)$, once again assuming that the I/Os can be parallelized, and the cost of partitioning for each iteration is negligible. I/O costs (these also represent the insertion and fetch costs into temporary tables that stores runs) need also to be minimized in a database system. The second partitioning method is to use a sampling method that divides the sequences based on a sort key (QUI88, see section 1). These methods perform a p-way merge of the partitions leading to a theoretical $O((N \log p)/p)$ for the expected number of comparisons per processor and $O(N/p)$ for number or I/Os per processors. However, since the original partitioning was only approximate, any skew in key values will lead to workload imbalance and a degradation in performance. In fact, Quinn (QUI88, see section 1) indicates that even with randomly distributed data, no speedup is obtained beyond $p = 10$ because of this workload imbalance.

In contrast, our algorithm will find an arbitrary exact partition for an arbitrary number of sequences. This leads to a nearly perfect workload balance and permits a single p-way merge phase. Thus the order of the merge is $O((N \log p)/p)$ for comparisons per processor and $O(N/p)$ for I/Os per processor, and solves the load balancing problem in parallel multi way merge. It is currently the best approach of which we are aware that will provide linear speedup in merge with respect to number of processors independent of key value distributions. Currently, implementations are underway to obtain empirical performance figures, and to optimize performance of the overall parallel sort by determining the optimal number of runs to merge in parallel given a fixed number of processors, I/O subsystem, and file organization.

5 Conclusions and Implementation Notes In this paper we examined the problems involved in employing multiple processors for parallel execution of a frequently used relational database function: external sorting. Previous solutions found in the database literature either rely on two way merges that makes the cost of merging expensive or use a final sequential merge that adversely impacts the speedup. We pick the best among three previously implemented sort algorithms for a shared memory parallel computer. Speedup due to this algorithm was shown to peak at about 10 processors. It failed to achieve efficiency and linear speedup beyond 10 processors because of approximations in the algorithm used to partition multiple sorted runs into equal percentile ranges. The approximations caused load imbalances among the

multiple processors. We have given an algorithm that can solve the percentile finding problem exactly that works for an arbitrary number of runs and arbitrary percentiles eliminating the possibility of load imbalance. By employing our algorithm in parallel external sorting and eliminating load imbalances due to the previous approximations linear speed up is possible for the number of processors typically used for database related operations. We have also given an example to illustrate the working of our percentile finding algorithm. The percentile finding algorithm has been prototyped and tested successfully on various sets of sorted runs. A comparison between our algorithm and Quinn's parallel sorting method using random and skewed key values is necessary to determine the actual benefits provided by each approach.

From an implementation viewpoint we need to address the question of how to directly access an arbitrary *i*'th row of a run, as required by our percentile finding algorithm. The issue is for variable length rows. We can either use an index or estimate the location location based on an average row size. We suggest an alternate approach: to place load balancing requirements (partition size requirements) in terms of number of pages of rows merged by each processor, rather than in number or rows as per the exposition in this paper. The algorithm given in this paper can be extended to solve the run partitioning problem with such a load balancing requirement.

Acknowledgement We acknowledge help from Pat Selinger, Don Haderle, Dennis DeLorme, John Vriezen, John Palmer and Joel Wolf.

References

(AKL87) Akl, S. G., and Santroo, N., "Optimal Parallel Merging and Sorting Without Memory Conflicts" IEEE Trans. on Comp., C36, 11, 1987, 1367-69.

(BAT68) Batcher, K. E., "Sorting Networks and their Applications", Proc. AFIPS 1968 Spring Jt. Computer Conf., Vol 32, AFIPS Press, Arlington, VA.

(BEC88) Beck, M., Bitton, D. and Wilkinson, W. K., "Sorting Large Files on a Backend Multiprocessor", IEEE Transactions on Computers, 37, 7, July 1988.

(BIT83) Bitton, D., Boral, H., DeWitt, D. J., and Wilkinson, W. K., "Parallel Algorithms for Relational Database Operations", ACM Transactions on Database Systems, Vol. 8, No. 3, Sept. 1983

(DEO88) Deo, N. and Sarkar, D., "Optimal Parallel Algorithms for Merging and Sorting", Proc. Third Intl. Conf. on Supercomputing, Boston, May 15-20, 1988, 513-521.

(FRA88) Frances, R. S. and Mathieson, I. D., "A Benchmark Parallel Sort for Shared Memory Multiprocessors", IEEE Trans. on Computers, Vol C37, No. 12, Dec. 1988, pp. 1619-26.

(FRE82) Fredrickson, G. N. and Johnson, D. B., "The Complexity of Selection and Ranking in X + Y and Matrices with Sorted Columns:, J. Comp. and Sys. Sciences, 24, 197-208, 1982.

(GRA86) Gray, J., Stewart, M., Tsukerman, A., Uren, S. and Vaughen, B., "Fastsort: An external sort using Parallel Processing", Tandem Syst. Review, Vol. 2, Dec. 1986.

(IYE88) Iyer, B. R. and Dias, D. M., "System Issues in Parallel Sorting for Database Systems", IBM Research Report RJ 6585, San Jose, CA, November 30, 1988.

(IYE89) Iyer, B. R., Ricard, G. R. and Varman, P. J., "An Efficient Percentile Finding Algorithm for Multiple Sorted Runs", IBM Research Report, San Jose, CA, 1989.

(LOR88) Lorie, R. A. and Young, H. C., "A Low Communication Sort Algorithm for a Parallel Database Machine", IBM Research Report RJ 6669, San Jose, CA, February, 1989 (also to appear in VLDB 89).

(QUI88) Quinn, M. J., "Parallel sorting algorithms for tightly coupled multi-processors", Parallel Computing, 6, 1988, pp. 349-367.

(KNU73) Knuth, D. E., "The Art of Computer Programming: Volume 3", Addison-Wesley, Reading, MA, 1973.

(OLK86) Olken, F., Rotem, D., "Simple Random Sampling from Relational Databases", Proc. of the 12'th Intl. Conf. on Very Large Data Bases, Kyoto, Japan, August 1986, pp. 160-169.

(VAL84) Valduriez, P., Gardarin, G., "Join and Semijoin Algorithms for a Multiprocessor Database Machine", ACM Trans. on Data Base Systems, Vol. 9, No. 1, March 1984, pp. 133-161.

(VAR88) Varman, P. J., Iyer, B. R. and Haderle, D. J., "Parallel Merge on an arbitrary number of Processors", IBM Internal Report, August 1988 (available as IBM Research Report RJ 6632, San Jose, CA, December 30, 1988).

(VIT85) Vitter, J. F, "Random Sampling with a Reservoir", ACM Trans. on Math. Software, Vol. 11, No. 1, March 1985, pp. 37-57.

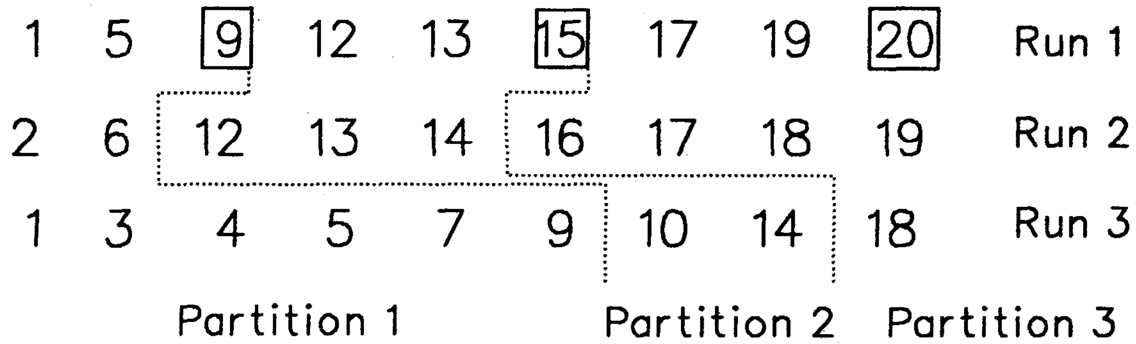

Figure 1

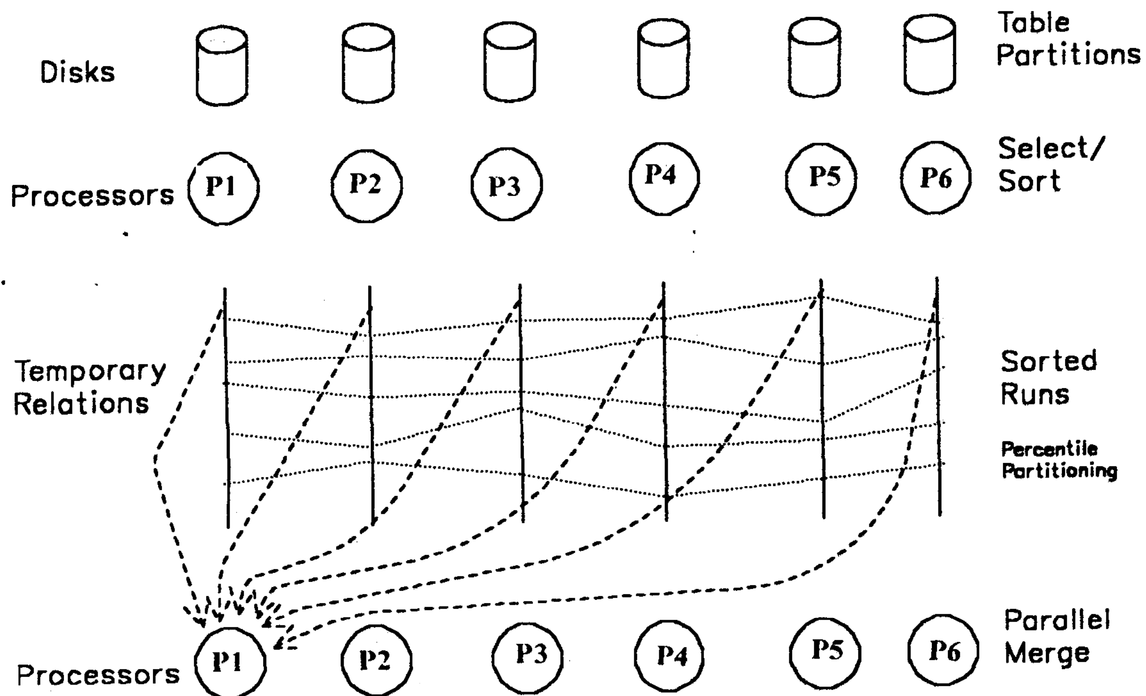

Figure 2