A Parallel Algorithm for Record Clustering

EDWARD OMIECINSKI
Georgia Institute of Technology
and
PETER SCHEUERMANN
Northwestern University

We present an efficient heuristic algorithm for record clustering that can run on a SIMD machine. We introduce the P-tree, and its associated numbering scheme, which in the split phase allows each processor independently to compute the unique cluster number of a record satisfying an arbitrary query. We show that by restricting ourselves in the merge phase to combining only sibling clusters, we obtain a parallel algorithm whose speedup ratio is optimal in the number of processors used. Finally, we report on experiments showing that our method produces substantial savings in an environment with relatively little overlap among the queries.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*single-instruction-stream, multiple-data-stream processors (SIMD)*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sequencing and scheduling*; H.3.2 [**Information Storage and Retrieval**]: Information, Storage; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Algorithms, Experimentation, Performance

1. INTRODUCTION

The availability of parallel computers has stimulated much interest in finding parallel algorithms in a number of areas, such as sorting [3, 4, 8] and graph theoretic problems [17, 18]. In the area of database systems, research into parallel algorithms has been basically limited to performing relational database operations, such as the join in parallel [5, 21], and to allocating data to independently accessible disks [6]. In this paper, we show that record clustering is another problem that possesses intrinsic parallelism, and we introduce an efficient parallel algorithm for a SIMD machine.

Both record clustering and attribute partitioning have the same objective of minimizing the number of accesses to secondary storage by placing the records (or attributes) most likely to be referenced together in the same queries into

Authors' addresses: E. Omiecinski, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332; P. Scheuermann, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208-3118.

related clusters (or subfiles). However, the techniques applied to attribute partitioning, such as the Bond Energy algorithm [20], are too expensive for record clustering, since the relevant input size for record clustering (the number of records) is several orders of magnitude larger than the input size for attribute partitioning (the number of attributes).

On the other hand, record clustering can be viewed as a complementary problem to indexing. The performance of a file organization dealing with complex queries can be improved substantially by providing an appropriate directory or index which will identify those records that satisfy a given query [19]. However, if the data records themselves are not clustered in the file, they could span many pages [22]. Thus, an appropriate index will reduce the number of records to be retrieved, while clustering related records on the same, or adjacent pages, will guarantee that most of the time their retrieval will not require separate accesses to the data file.

In the past few years, a number of dynamic, multiattribute file organizations have been proposed that do not require knowledge of statistics about a file's use and can perform an incremental restructuring that maintains high performance when dealing with insertions and deletions [11, 19]. These schemes can be viewed as Cartesian product clustering methods. They partition either the embedding space, or the space of the specific set of stored records, into cells that correspond to buckets (pages) on secondary devices. However, as with most other clustering algorithms [10], these methods are effective for conjunctive queries only.

In [12, 14] we have introduced an approach to record-clustering and companion file reorganization that can be applied to all kinds of queries. The clustering algorithm, called SPLITMERGE, assumes that for each relevant query we keep statistics on its frequency of occurrence as well as the identifiers (addresses) of the records that satisfy the query. The output of the clustering algorithm is a list of clusters containing the records pertinent to different subsets of queries. As shown in [13, 23], the problem of determining an optimal clustering of records is NP-hard; hence our approach is a heuristic one, which produces a near-optimal list of clusters.

The SPLITMERGE algorithm has the advantage of being applicable to all kinds of queries, not just conjunctive ones. On the other hand, it has a worst-case time complexity of $O(M * N)$ and a space complexity of $O(N)$, where M is the number of relevant queries and N is the number of records. We present an efficient, parallel clustering algorithm that is based on the sequential SPLITMERGE. Our algorithm is intended for a SIMD (Single Instruction stream, Multiple Data stream) machine having K processors, with local memory interconnected by an interconnection network. As is the case for parallel external sorting algorithms [4], we also assume that the number of processors is sublinear with respect to the number of records in the file; that is, $K <= \log_2 N$. The parallel algorithm has a time complexity of $O(M * N/K + N)$, and its speedup ratio is optimal in the number of processors used. In addition, we relax the requirement that the record identifiers that satisfy a query must be available in advance and show how each processor can compute independently for a given record a unique cluster number that indicates the queries it satisfies using only $O(N/K)$ space.

This paper is organized as follows. In Section 2 we formulate the problem and review the SPLITMERGE clustering algorithm. Section 3 describes the P-tree

and its associated numbering scheme, which introduces the potential for parallelism in the cluster identification process. The PARALLEL SPLITMERGE algorithm is presented and analyzed for correctness and complexity in Section 4. Section 5 reports on a number of synthetic experiments we carried out to evaluate the performance of our algorithm in terms of the quality of the clustering produced.

2. THE SERIAL ALGORITHM

For our record-clustering problem we considered a file of N records and a set of M queries, $\{Q_1, \ldots, Q_M\}$, to be processed against the file. We assumed that for each query the following statistics are available: its frequency of occurrence, and the identifiers (addresses) of the records which satisfy the query. Actually, $\{Q_1, \ldots, Q_M\}$ is a subset of Q^*, the set of all possible queries containing the most frequently requested queries. The objective function we want to minimize is:

$$C = \sum_{i=1}^{M} F(Q_i) * P(Q_i) \qquad \begin{array}{l}\text{Where } F(Q_i) = \text{Frequency of query } Q_i \\ \text{and } P(Q_i) = \text{number of pages which} \\ \text{contain records for the query } Q_i\end{array} \qquad (1)$$

The serial clustering algorithm, SPLITMERGE, consists of two components: the logical phase and the physical phase. Basically, in the logical phase, records are assigned to clusters such that records that satisfy a given subset of $\{Q_1, Q_2, \ldots, Q_M\}$ end up in the same cluster. Assume that we are processing the queries in some order $Q_{i_1}, Q_{i_2}, \ldots, Q_{i_M}$. When we consider query Q_{i_j}, we will group together the records that satisfy this query but do not satisfy any of the preceding queries $\{Q_{i_1}, Q_{i_2}, \ldots, Q_{i_{j-1}}\}$, in order to reduce $P(Q_{i_j})$. On the other hand, if there are some records that satisfy Q_{i_j} as well as a subset of $\{Q_{i_1}, Q_{i_2}, \ldots, Q_{i_{j-1}}\}$, an existing cluster is split into two parts such that the intersecting set of records creates a separate cluster.

Let R_{Q_i} be the set of records that satisfy Q_i and $r.a$ denote the identifier (or address) of record r. We also denote by $\bar{R}_{Q_i}$ the set of record identifiers that satisfy Q_i, that is, $\bar{R}_{Q_i} = \{r.a \mid r \in R_{Qi}\}$. The SPLITMERGE algorithm produces clusters of record identifiers that satisfy only a single query Q_i, denoted by $\bar{Q}_i$, as well as clusters that satisfy Q_i and another subset s of queries, where $s \subseteq \{1, \ldots, M\} - \{i\}$, denoted by $\bar{Q}_{is}$. We shall refer to $\bar{Q}_i$ as a primary cluster of Q_i and any $\bar{Q}_{is}$ as a secondary cluster of Q_i.

In the physical phase, we produce a mapping of clusters to fixed-size pages. For each query Q_i, we attempt to assign to the same page, or neighboring pages, not only the records belonging to the primary cluster $\bar{Q}_i$, but also those belonging to its secondary clusters $\bar{Q}_{is}$. But since $\bar{Q}_{is} = \bar{Q}_{\Pi(is)}$, where Π stands for a permutation function, and the assignment to storage is done in a nonredundant fashion, we must decide which secondary clusters get materialized and which do not.

If a query occurs very often, we want to minimize the number of pages accessed for that query. However, we must also consider the query set size, that is, the number of records satisfying a query, since the number of pages accessed when records are randomly placed approaches the query set size as shown in Yao [22]. To account for both these factors in the minimization of the objective function

(1), we arrange the queries in the following order: $Q_i \alpha Q_j$ (Q_i precedes Q_j) if and only if $F(Q_i) * S(Q_i) \geq F(Q_j) * S(Q_j)$, where $S(Q_i)$, the query set size for Q_i, is defined as the cardinality of R_{Q_i}. The logical phase of SPLITMERGE processes the queries in this order, and as a result, only the secondary clusters whose index satisfies this precedence order are materialized, that is,

$$\bar{Q}^n_{i_1,i_2,\ldots,i_j} = \begin{cases} \{r.a \mid r \in R_{Q_{i_k}} \text{ and } r \notin R_{Q_{i_l}}\} \\ \qquad \text{where } i_k \in \{i_1, \ldots, i_j\} \text{ and } i_l \in = \{i_1, \ldots, i_n\} - \{i_1, \ldots, i_j\} \\ \quad \text{if } Q_{i_1} \alpha Q_{i_2} \alpha \cdots \alpha Q_{i_j} \\ 0 \quad \text{otherwise} \end{cases}$$

The superscript n in $\bar{Q}^n_{i_1,i_2,\ldots,i_j}$, for $1 \leq n \leq M$, denotes the position (order) of Q_{i_n}, the last query in the precedence order for which the logical phase of SPLITMERGE was completed. Note that at the end of the logical phase (that is, when $n = M$), the cluster $\bar{Q}^M_{i_1,i_2,\ldots,i_j}$ indeed contains only identifiers of the records satisfying $Q_{i_1}, Q_{i_2}, \ldots, Q_{i_j}$. Thus, we obtain $\bar{Q}_i = \bar{Q}^M_i$ (and similarly $\bar{Q}_{is} = \bar{Q}^M_{is}$). On the other hand, during each iteration of the logical phase, an existing cluster $\bar{Q}^n_{i_1,i_2,\ldots,i_j}$ will be split into $\bar{Q}^{n+1}_{i_1,i_2,\ldots,i_j,i_{n+1}}$ and $\bar{Q}^{n+1}_{i_1,i_2,\ldots,i_j}$ if it has records in common with $R_{Q_{n+1}}$. However, this step is executed only if the size of $\bar{Q}^n_{i_1,i_2,\ldots,i_j}$ exceeds the physical page size, as our objective is to group related records on the same physical page, if possible.

In addition to splitting an existing cluster, SPLITMERGE also incorporates a merge step for every query Q_n. This is done to avoid the possibility of small secondary clusters, $\bar{Q}^{n-1}_{js_1}, \ldots, \bar{Q}^{n-1}_{js_k}$ for $j < n$, being assigned to k different pages in the physical phase. Since Q_j precedes Q_n, earlier split steps are not able to detect whether these clusters are needed for answering Q_n. If during iteration n we find that they are required, and their combined size does not exceed the physical page size, we merge them.

The result of the physical phase of SPLITMERGE can be seen as a mapping Δ of N record identifiers to S pages of storage [12, 13]. The actual transformation of the file from the old state corresponding to the pre-clustering configuration, described by a mapping Δ_{old}, to the new state, defined by Δ, is accomplished by a separate reorganization algorithm. In [12] and [14] we have introduced efficient incremental reorganization algorithms that allow for concurrent reorganization with user access to the file.

As we stated earlier, a record-clustering approach can be viewed as complementing a directory structure that provides an index(es) to the different attributes appearing in the set of queries $Q_1, \ldots, Q_M$. In order to perform the reorganization efficiently, a page table PG can be used [9], which associates with every record identifier the page number on which it resides. Thus, between the directory and the data file, we introduce an additional level of indirection; the accession pointers in the directory [19] are not pointing to the data file anymore, but to the page table PG. This scheme has the advantage that any changes to the data file affect only the PG, and not the directory.

Alternatively, if no index(es) on the attributes appearing in the queries are kept, we can construct a clustered index [10] in the physical phase of SPLITMERGE in order to identify for each query Q_i where its relevant records are

stored. We observe that such a clustered index is much smaller than an index of the various attributes in the queries. In particular, it suffices to store for each query Q_i a beginning and end address of the pages, where its primary cluster $\bar{Q}_i$ and secondary clusters $\bar{Q}_{is}$ are stored and a similar list of addresses for the secondary clusters $Q_{\Pi(is)}$, which exist, but for which $\bar{Q}_{is}$ is not materialized.

3. THE PARTITION TREE AND ITS NUMBERING SCHEME

In this section we describe the Partition Tree (P-tree) and the associated P-tree-numbering scheme, which form the basis for the parallel algorithm. By making no restrictions a priori on the cluster sizes in the split step, we introduce the potential for parallelism in the cluster decomposition process.

The partition tree, like a decision tree, represents all possible clusters, including empty ones, that can be formed from a file by examining a given set of queries. The underlying structure of a P-tree is a binary tree, in which each level corresponds to a different query. After M queries have been processed against a file, the external nodes correspond to all the distinct clusters involving the given M queries.

We associate with each node of the P-tree a cluster number. Each record, in turn, is now mapped to the unique cluster number of an external node in the following way. As the record is examined to determine whether or not it satisfies a given query, a bit is set to on/off in a fixed position corresponding to this query in the binary representation of the cluster number. This process is repeated for all queries, but the inherent parallelism is due to the fact that we can assign a cluster number to a record (on one processor) independently of assigning a cluster number to a different record (on any other processor). In addition, the P-tree-numbering scheme guarantees that a minimal amount of interprocessor communications is required to determine whether the resulting clusters should be merged or not. We now proceed to define more formally the P-tree and its associated numbering scheme.

Definition 1. The Partition tree (P-tree) of a file F is a binary tree of height h representing all the clusters, including possibly empty ones, which are induced by $h - 1$ queries. Thus, the nodes at level i ($i \leq h - 1$) represent the clusters formed after examining the first i queries in the given precedence order. Let N^i be an arbitrary node at level i in this full binary tree. We associate with each N^i a set of record identifiers $\bar{R}(N^i)$. The structure of the P-tree and the corresponding $\bar{R}(N^i)$ are defined as follows:

(1) The root node N^0 always exists and $\bar{R}(N^0) = \bar{F}$ where $\bar{F}$ is the set of all record identifiers in F.
(2) If $\bar{R}(N^{i-1}) \neq 0$, then the children of N^{i-1} exist and
   (a) $\bar{R}(LEFTCHILD(N^{i-1})) = \bar{R}(N^{i-1}) \cap \bar{R}_{Q_i}\ 1 \leq i < h$ and
   (b) $\bar{R}(RIGHTCHILD(N^{i-1})) = \bar{R}(N^{i-1}) - \bar{R}_{Q_i}\ 1 \leq i < h$.

(*Note*: Strictly speaking, the structure of the P-tree corresponds to a full binary tree, but for simplicity, we elect not to extend further empty nodes.)

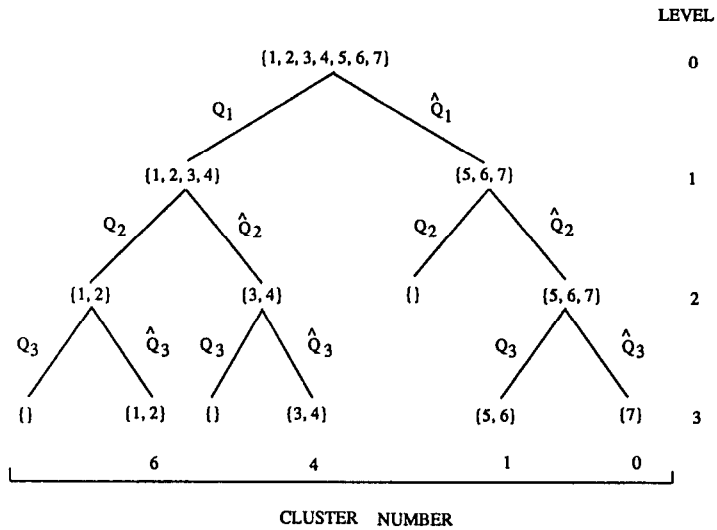


Fig. 1. Partition tree.

As an example, let us consider the file $\bar{F} = \{1, 2, 3, 4, 5, 6, 7\}$ and the queries $Q_1 \alpha Q_2 \alpha Q_3$ for which we have: $\bar{R}_{Q_1} = \{1, 2, 3, 4\}$, $\bar{R}_{Q_2} = \{1, 2\}$, and $\bar{R}_{Q_3} = \{5, 6\}$. The corresponding P-tree of height 4 is illustrated in Figure 1, in which next to each node N^i is shown its set $\bar{R}(N^i)$.

At level 0, no queries have been processed, and the set $\bar{R}(N^0)$ contains all the record identifiers in the file. At level 1, after Q_1 has been processed, there exist two clusters (that is, $\bar{R}(LEFTCHILD(N^0)) = \bar{R}(N^0) \cap \bar{R}_{Q_1} = \{1, 2, 3, 4\}$ and $\bar{R}(RIGHTCHILD(N^0)) = \bar{R}(N^0) - \bar{R}_{Q_1} = \{5, 6, 7\}$). Finally, after all three queries have proceeded, seven possible clusters are identified, out of which only four, corresponding to the paths $Q_1Q_2\hat{Q}_3$, $Q_1\hat{Q}_2\hat{Q}_3$, $\hat{Q}_1Q_2Q_3$, and $\hat{Q}_1\hat{Q}_2\hat{Q}_3$, are nonempty.

We now superimpose on the P-tree a numbering scheme that will enable us to assign independently to each record identifier a cluster number. Let us assume the P-tree of a file has h levels corresponding to $h - 1$ queries. With query Q_i in the given precedence order, we associate the base cluster number 2^{h-1-i}. Initially, all record identifiers are assigned 0 as their cluster number. At level i in the P-tree, for all record identifiers in $\bar{R}_{Q_i}$, we add the base number 2^{h-1-i} to their previous cluster numbers. Thus, the P-tree-numbering scheme can be described as given below.

Definition 2. The P-tree-numbering scheme assigns a cluster number, denoted by CLUSTER-NO(N^i), to each node N^i at level i in the P-tree as follows:

(1) CLUSTER-NO(N^0) = 0.

(2) If N^{i-1} has children then

CLUSTER-NO(LEFTCHILD(N^{i-1})) = CLUSTER-NO(N^{i-1}) + 2^{h-1-i}

CLUSTER-NO(RIGHTCHILD(N^{i-1})) = CLUSTER-NO(N^{i-1})

for $1 \leq i < h$.

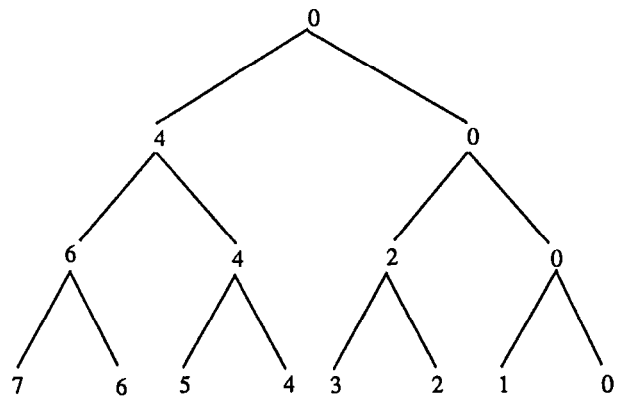


Fig. 2. P-tree-numbering scheme.

Figure 2 illustrates the P-tree-numbering scheme on a full binary tree of height 4. Correspondingly, the nonempty external nodes of the P-tree in Figure 1 correspond to clusters numbered 0, 1, 4, and 6. Let us denote with CLUSTER-NO(L^i) the set of cluster numbers associated with nodes at level i:

$$\text{CLUSTER-NO}(L^i) = \{\text{CLUSTER-NO}(N^i) \mid N^i \text{ is a node at level } i\}.$$

From Definition 2, we obtain the following equality:

$$\text{CLUSTER-NO}(L^i) = \text{CLUSTER-NO}(L^{i-1}) \cup \{j + 2^{h-1-i} \mid j \in \text{CLUSTER-NO}(L^{i-1})\}.$$

Furthermore, for the last level in the P-tree, that is, for $i = h - 1$, our numbering scheme implies the following:

PROPERTY 1. *Sibling leaves in the P-tree are assigned consecutive cluster numbers.*

An additional property of our numbering scheme will be used in the parallel clustering algorithm.

PROPERTY 2. *Given the cluster number C_r of a node in a P-tree at level i, then the cluster number of its parent is $\lfloor C_r/2^{h-i} \rfloor * 2^{h-i}$, where h is the height of the P-tree.*

PROOF. From Definition 2, we know that if CLUSTER-NO(N^{i-1}) = j, then the cluster number of its right child (if it exists) is j, and the cluster number of its left child (if it exists) is $j + 2^{h-1-i}$. Hence, we have to show that if we apply the above formula to the cluster number of either child, we obtain j; that is,

$$\lfloor j/2^{h-i} \rfloor * 2^{h-i} = \lfloor j + 2^{h-1-i}/2^{h-i} \rfloor * 2^{h-i} = j.$$

The preceding expression is equivalent to:

$$\lfloor j/2^{h-i} \rfloor = \lfloor j/2^{h-i} + \tfrac{1}{2} \rfloor = j/2^{h-i}.$$

It can easily be shown by induction that j is divisible by 2^{h-i} [13], and hence the above equality is established. □

4. PARALLEL SPLITMERGE

Our parallel clustering algorithm is designed for a SIMD (Single Instruction stream, Multiple Data stream) machine having K processors, each with its own memory and linked by an interconnection network. As in [3], we assume the existence of a route instruction that allows a given processor to read a memory location of an adjacent processor and store the contents in its own memory. The interconnection scheme is a linear order, that is, processor P^i is adjacent to P^j where $j = i - 1 \,(\text{mod K})$ or $j = i + 1 \,(\text{mod K})$. All processors are synchronized, but each processor has the capability of inhibiting the execution of the current instruction. As is the case in parallel external-sorting algorithms for large files [4, 15], we assume that the number of processors is sublinear with respect to the number of records in the file; that is, $K \leq \log_2 N$. We note that, in general, faster parallel algorithms can be obtained for SIMD machines having a shared memory, but this requires a number of processors that are linear, at least in the input size [4, 8], an assumption that is not very realistic for database systems.

The parallel heuristic algorithm for the above SIMD model consists of four modules: SPLIT, SORT, and MERGE, which implement the logical phase, and ALLOCATE, which accomplishes the physical phase. Since the ALLOCATE procedure is a serial allocation scheme similar to the one presented in [12], we shall not discuss it further. The SPLIT and MERGE procedures are based on the P-tree and its numbering scheme, while the SORT procedure implements the parallel sorting algorithm developed in [3] for an SIMD machine with a mesh interconnection scheme.

In the SPLIT phase of the parallel algorithm, we do not halt the splitting of a cluster if its size is less than or equal to the page size because the records forming a cluster may be distributed among a number of processors, and the communications cost to determine the cluster sizes would be too great. The SORT and MERGE procedures are designed to remedy this problem, that is, to merge clusters back to the page size.

As input to the SPLIT algorithm we have a file F that is being read in parallel by the K processors. One suitable hardware design would be a modified moving head disk that provides for parallel read/write [2]. Each processor reads a portion F^j of the file and computes the vectors REC-ID^j and CLUSTER-NO^j, containing the identifiers of the records in F^j and their cluster numbers. Specifically, processor P^j will store in REC-ID^j[1:N/K] the identifiers of the records in positions $(j - 1) * N/K + 1$ through $j * N/K$ in the file and the corresponding cluster numbers in CLUSTER-NO^j[1:N/K]. (We assume for simplicity that

N/K is an integer.) The SPLIT algorithm is outlined below:

Algorithm 1. SPLIT

```
Input:   F^j, Q_1 ∝ Q_2 α ··· α Q_M
Output:  REC-ID^j[1:N/K]
         CLUSTER-NO^j[1:N/K]
For each F^j do in parallel
  For p = 1 to N/K do
      /* Initialization steps */
    Read F^j(p);
      CLUSTER-NO^j[p] = 0;
      REC-ID^j[p] = (j − 1) * N/K + p;
    For i = 1 to M do /* once for each query */
      If F^j[p] satisfies Q_i
        then CLUSTER-NO^j[p] = CLUSTER-NO^j[p] + 2^(M−i)
    end
  end
end
```

Let us consider the following example:

Example 1. $\bar{F} = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $\bar{R}_{Q_1} = \{1, 3, 5, 7\}$, $\bar{R}_{Q_2} = \{1, 2, 5, 6\}$, $K = 2$, and PAGESIZE = 4.

The corresponding P-tree and the corresponding cluster numbers for this file are illustrated in Figure 3a.

After the execution of the SPLIT algorithm, the configuration of the processors's memory is

P^1: REC-ID^1[1:4] = 1, 2, 3, 4
CLUSTER-NO^1[1:4] = 3, 1, 2, 0

P^2: REC-ID^2[1:4] = 5, 6, 7, 8
CLUSTER-NO^2[1:4] = 3, 1, 2, 0

Since we did not check the size of the clusters during the split process, we ended up with four clusters, all smaller than the PAGESIZE, which, in this case, equals four. To remedy this situation, we merge the sibling clusters to obtain the revised P-tree shown in Figure 3b.

The basic idea behind the MERGE algorithm is similar to that of the buddy system [1] for memory management. We start with the original clusters corresponding to the nodes at the highest level in the P-tree and combine siblings if their combined size does not exceed the page size. This step is repeated for each level, up to level 0, or until no more clusters can be merged.

The reason for restricting ourselves to merging sibling clusters only is to minimize the amount of interprocessor communication required. By performing a parallel sort algorithm, we can reassign the record identifiers to the processors' memories based on a nonincreasing order of cluster numbers. Thus, by sorting the list of cluster numbers of the N records, the record identifiers with the largest N/K cluster numbers will be reassigned to the memory of P^1, the identifiers with the next largest N/K cluster numbers will be reassigned to the memory of P^2, and so on. The result of a parallel sort on the configuration of Example 1 results

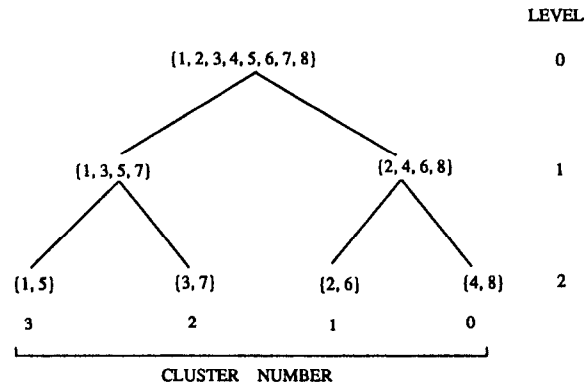


Fig. 3a. P-tree for Example 1.

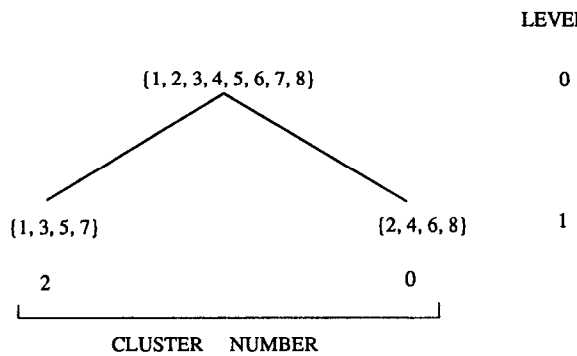


Fig. 3b. Revised P-tree.

in the following memory assignment for the two processors:

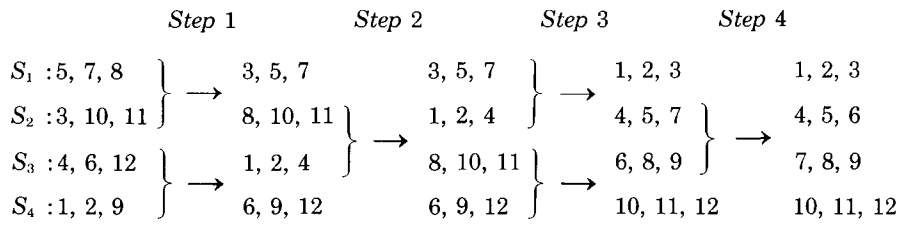
P^1: REC-ID^1[1:4] = 1, 5, 3, 7
CLUSTER-NO^1[1:4] = 3, 3, 3, 2

P^2: REC-ID^2[1:4] = 2, 6, 4, 8
CLUSTER-NO^2[1:4] = 1, 1, 0, 0

As a result of Property 1, shown in Section 2, it follows that after the sort is executed, the record identifiers for sibling clusters are in consecutive memory locations of a processor, except for the situation in which a cluster may span the memory of two or more adjacent processors. Thus, as we shall see in more detail later, we can limit the amount of communication needed for routing instructions between adjacent processors.

For the second step of our parallel clustering algorithm, we employ the parallel sorting algorithm developed by Baudet and Stevenson [3]. Their algorithm is basically a generalization of the odd–even transposition sort. Let us consider a partially sorted sequence of numbers, $S = S_1, S_2, S_3, \ldots, S_K$. Each S_i subsequence, consisting of r elements and stored in the memory of processor P^i, is sorted, but the whole sequence S is not. During an odd step, processors P^i for $i = 1, 3, 5, \ldots$ are active; they merge the two subsequences S_i and S_{i+1} and then assign to S_i the

first half of the resulting merged sequence (that is, the r smallest elements) and to S_{i+1} the second half (that is, the r largest elements). During an even step, processors P^i for $i = 2, 4, 6, \ldots$ are active, and they repeat the same operations. An example of the parallel sorting algorithm is shown below.



The details of the MERGE algorithm and its subordinated procedure, SPANNING, are illustrated below. To do the merging, we must first determine the parent cluster numbers for each cluster, using Property 2 shown in Section 2 (Step 2.1). Next, we check that the combined size of sibling clusters is not greater than the page size by partitioning the vector PARENT-NO^j[1:N/K] into sets with an identical parent cluster number; that is, for each distinct p in the parent vector, we compute COMMON-PARENT^j(p) (Step 2.3). If the size of a partition COMMON-PARENT^j(p) is smaller than, or equal to, the page size, the merge of the descending sibling clusters is recognized; otherwise, the merge is undone by restoring the cluster numbers to their previous values (Step 2.5). This checking suffices if the parent cluster does not span adjacent processors. In addition, each processor computes FIRST^j, FIRSTSIZE^j, LAST^j, and LASTSIZE^j (Step 2.2). FIRST^j and LAST^j are the highest and lowest parent cluster numbers in PARENT-NO^j[1:N/K]. FIRSTSIZE^j and LASTSIZE^j denote the size of the clusters numbered FIRST^j and LAST^j. The SPANNING procedure is invoked (Step 2.4) to check whether the records of a parent cluster span adjacent processors. Each processor P^j must check whether a parent cluster spans P^j and P^{j-1} and computes its combined size and, in addition, checks whether another parent cluster spans P^j and P^{j+1} and repeats the same procedure. If the size of a parent cluster exceeds the page size, the cluster numbers must be reset to their previous values.

We proceed now with a more detailed description of SPANNING. The SPANNING procedure for processor P^j begins by reading from the memory of processor P^{j-1} the variables LAST^{j-1} and LASTSIZE^{j-1} and from P^{j+1} the variables FIRST^{j+1} and FIRSTSIZE^{j+1}. SPANNING then compares FIRST^j and LAST^{j-1} (Step 2). If they are equal, this indicates that the cluster spans processors P^j and P^{j-1}, and the combined size (that is, FIRSTSIZE^j and LASTSIZE^{j-1}) must be compared to the page size. If the combined size is not greater than the page size, then it is a valid cluster, and no further action is necessary. When the sum equals the page size, we know that this cluster should not be combined with its sibling in a successive step. To prevent further combining in later steps, the FIRST-SPAN-FLAG^j is set to false (Step 2.1), and FLAG^j is also set to false for each individual record contained in the cluster. These flags are also set in the case when the sum is greater than the page size. When the sum FIRSTSIZE^j + LASTSIZE^{j-1} exceeds the page size, then the proposed

cluster is too large. Hence, the two previous cluster numbers (that is, the sibling numbers) must be restored (Step 2.2). The PARENT-NO^j[i] number, for each record i in the cluster being examined, is set to CLUSTER-NO^j[i]. This is necessary, since in Step 2.5 of MERGE, the CLUSTER-NO^j vector is updated from the PARENT-NO^j vector to reflect the modified clusters (that is, the merging of two clusters into one). The same process is repeated for the right neighbor; that is, P^j compares LAST^j with FIRST^{j+1}, and so on (Step 3). In addition, if we find that the FIRST^j and LAST^j clusters on P^j cannot be merged any further with clusters on P^{j-1} and P^{j+1}, respectively, then SPAN-FLAG^j is set to false (Step 4). A value of false for SPAN-FLAG^j would inhibit the calling of SPANNING for processor P^j from the MERGE procedure.

Algorithm 2. MERGE

Input: CLUSTER-NO^j[1:N/K]
Output: Modified CLUSTER-NO^j[1:N/K]
Variables: PARENT-NO^j[1:N/K] = vector containing for each processor the cluster numbers of the parents in the P-tree corresponding to the entries in CLUSTER-NO^j[1:N/K]
COMMON-PARENT^j(p) = {i | PARENT-NO^j[i] = p}
FLAG^j[1:N/K] = Boolean vector; FLAG^j[p] becomes false when the cluster with number CLUSTER-NO^j[p] cannot be merged further
FIRST^j = the highest number among the parent cluster numbers in PARENT-NO^j[1:N/K]
LAST^j = the lowest number among the parent cluster numbers in PARENT-NO^j[1:N/K]
SPAN-FLAG^j = Boolean; if it is false, then the procedure SPANNING will not be called.
LASTSIZE^j = size of cluster whose number is LAST^j
FIRSTSIZE^j = size of cluster whose number is FIRST^j

Step 1: /* Initialization */
Set SPAN-FLAG^j, FLAG^j[1:N/K], FIRST-SPAN-FLAG^j, LAST-SPAN-FLAG^j to True in parallel;
$L \leftarrow M$ /* the height of the P-tree is $M + 1$ */
Step 2: While FLAG^j[1:N/K] ≠ False do in parallel
Step 2.1: For $r = 1$ to N/K do
PARENT-NO^j[r] = ⌊CLUSTER-NO^j[r]/2^{M+1-L}⌋ * 2^{M+1-L};
Step 2.2: Determine FIRST^j and LAST^j for each processor;
LASTSIZE^j = |{i | PARENT-NO^j[i] = LAST^j}|;
FIRSTSIZE^j = |{i | PARENT-NO^j[i] = FIRST^j}|;
Step 2.3: For each distinct p in PARENT-NO^j[1:N/K] do
700 Determine COMMON-PARENT^j(p);
(2.3.1): If there is an i in COMMON-PARENT^j(p) such that FLAG^j[i] = False
then do for each i in COMMON-PARENT^j(p)
Begin FLAG^j[i] = False;
PARENT^j[i] = CLUSTER-NO^j[i];
End
else Begin
(2.3.2): If |COMMON-PARENT^j(p)| ≥ PAGESIZE
then do for each i in COMMON-PARENT^j(p)
FLAG^j[i] = False;
(2.3.3): If |COMMON-PARENT^j(p)| > PAGESIZE
then do for each i in COMMON-PARENT^j(p)
PARENT^j[i] = CLUSTER-NO^j[i];
End;

Step 2.4: If SPAN-FLAG^j = True and $j < K$
then call SPANNING;
Step 2.5: /* Reset cluster number values for the next iteration */
For $i = 1$ to N/K do
CLUSTER-NO$^j[i]$ = PARENT-NO$^j[i]$;
Step 2.6: $L \leftarrow L - 1$;

Algorithm 3. SPANNING (for processor P^j)

Step 1: Read LAST^{j-1} and LASTSIZE^{j-1} (from processor P^{j-1}'s memory)
Read FIRST^{j+1} and FIRSTSIZE^{j+1} (from processor P^{j+1}'s memory)
Note 1: P^1 inhibits the execution of Step 2 and sets
FIRST-SPAN-FLAG^1 = False
Note 2: P^k inhibits the execution of Step 2 and sets
LAST-SPAN-FLAG^k = False
Step 2: If (LAST^{j-1} = FIRST^j) and FIRST-SPAN-FLAG^j then
Begin
Step 2.1: If LASTSIZE^{j-1} + FIRSTSIZE$^j \geq$ PAGESIZE then
Begin
For each i s.t. PARENT-NO$^j[i]$ = FIRST^j do
FLAG$^j[i]$ = False;
FIRST-SPAN-FLAG^j = False;
End;
Step 2.2: If LASTSIZE^{j-1} + FIRSTSIZE$^j >$ PAGESIZE then
For each i s.t. PARENT-NO$^j[i]$ = FIRST^j do
PARENT-NO$^j[i]$ = CLUSTER-NO$^j[i]$;
End;
Step 3: If (LAST^j = FIRST^{j+1}) and LAST-SPAN-FLAG^j then
Begin
Step 3.1: If LASTSIZE^j + FIRSTSIZE$^{j+1} \geq$ PAGESIZE then
Begin
For each i s.t. PARENT-NO$^j[i]$ = LAST^j do
FLAG$^j[i]$ = False;
LAST-SPAN-FLAG^j = False;
End;
Step 3.2: If LASTSIZE^j + FIRSTSIZE$^{j+1} >$ PAGESIZE then
For each i s.t. PARENT-NO$^j[i]$ = LAST^j do
PARENT-NO$^j[i]$ = CLUSTER-NO$^j[i]$;
End;
Step 4: SPAN-FLAG^j = FIRST-SPAN-FLAG^j OR LAST-SPAN-FLAG^j;

We now illustrate the interaction of the different modules of PARALLEL SPLITMERGE by looking at a complete example.

Example 2. We consider a file with 20 records for which $\bar{F} = 1, 2, \ldots, 20$, and a page size of 3 records. We utilize four processors and do the clustering with respect to the four queries listed below in nonincreasing order of frequency * query set size:

$$\bar{R}_{Q_1} = \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19\}$$
$$\bar{R}_{Q_2} = \{1, 2, 3, 4, 5, 6, 7\}$$
$$\bar{R}_{Q_3} = \{1, 3, 8, 9, 10, 11, 13, 18, 19\}$$
$$\bar{R}_{Q_4} = \{1, 2, 5, 9, 12, 14, 16\}$$

The configuration of the processors' memory after execution of the SPLIT phase is illustrated in Figure 4, and the corresponding P-tree is shown in

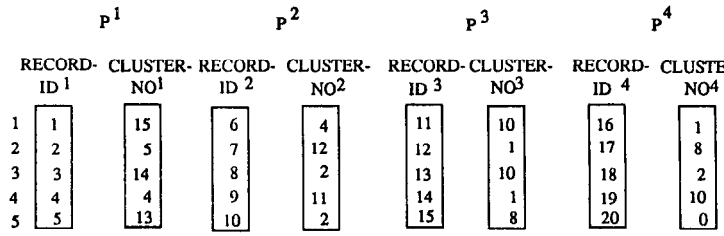


Fig. 4. Processor memory after SPLIT phase.

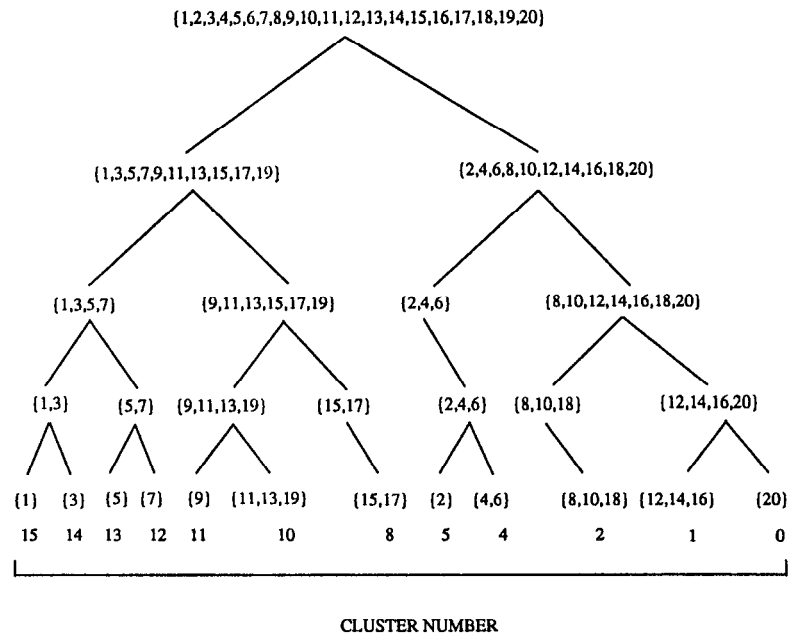


Fig. 5. P-tree.

Figure 5. At this point the parallel SORT algorithm is executed, resulting in the memory configuration of Figure 6.

We now enter the MERGE phase, and since the RECORD-ID^j vectors will not change further, we no longer exhibit them. We examine, however, the CLUSTER-NO^j, PARENT-NO^j, and FLAG^j vectors, as well as the FIRST^j, LAST^j, and SPAN-FLAG^j variables defined in the MERGE and SPANNING algorithms.

During the first iteration of MERGE, after steps 2.1 and 2.2 have been executed in parallel on all four processors, we have the configuration shown in Figure 7. During execution of step 2.3 on processor P^4, we find that the size of a parent cluster (that is, cluster 0) exceeds the page size. Hence, PARENT-NO^4 and FLAG^4 are changed. In addition, processors P^3 and P^2 contain parent clusters

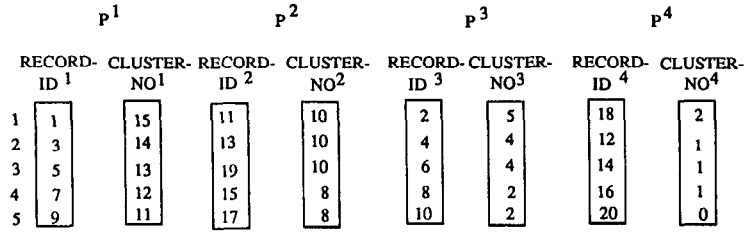


Fig. 6. Processor memory after SORTING.

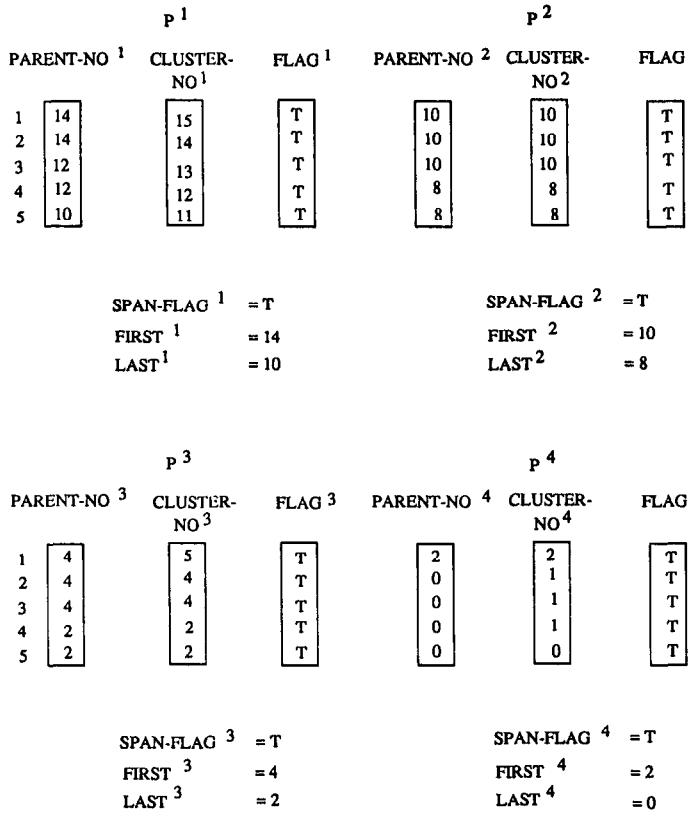


Fig. 7. Processor memory during first iteration of MERGE.

whose size equals the page size. To inhibit further attempts to merge these clusters during the next iterations, $FLAG^3$ and $FLAG^2$ are changed correspondingly. The changes made so far are depicted in Figure 8. During the execution of step 2.4, that is, the invocation of the SPANNING procedure, on processor P^1 we find the $LAST^1 = FIRST^2 = 10$, and the combined size of the cluster is 4, which exceeds the page size. Thus, the corresponding $FLAG^1$ and $FLAG^2$ entries,

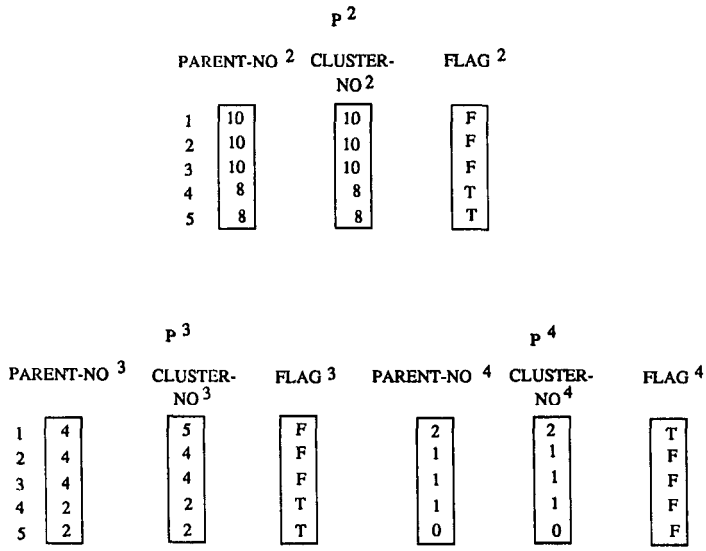


Fig. 8. Processor memory modifications.

as well as LAST-SPAN-FLAG[1] and FIRST-SPAN-FLAG[2], must be set to false. In addition, the entries in PARENT-NO[1] and PARENT-NO[2], corresponding to parent cluster number 10, are restored to the values of their original descendent cluster numbers. Since LAST[2] ≠ FIRST[3], nothing more is done for processor P^2. Procedure SPANNING invoked for processor P^3 finds LAST[3] = FIRST[4] = 2. In this case the combined size of the parent cluster equals the page size, so we just set the corresponding FLAG[3], FLAG[4] entries, and the LAST-SPAN-FLAG[3] and FIRST-SPAN-FLAG[4] variables to false. The result, after SPANNING is finished, appears in Figure 9.

For the second iteration of MERGE, we see that processors P^3 and P^4 will not execute step 2, since all FLAG[3] and FLAG[4] entries are set to false. After step 2.1 is executed for processors P^1 and P^2, we obtain the following changes, shown in Figure 10. During the execution of step 2.3 on processors P^1 and P^2, we find that parent clusters 12 and 8, respectively, exceed the page size, and PARENT-NO[1], FLAG[1], PARENT-NO[2], and FLAG[2] are changed as shown in Figure 11. Step 2.4 will not be executed on processor P^1, since SPAN-FLAG[1] is false. For processor P^2, SPANNING is called, but since LAST[2] ≠ FIRST[3], nothing is changed. After step 2.5 is executed, we are finished with the second iteration of MERGE. At this point all the entries in FLAG[1], FLAG[2], FLAG[3], and FLAG[4] are set to false. This condition terminates the execution on all processors. The final contents of the CLUSTER-NO[j] and REC-ID[j] vectors are displayed in Figure 12.

The space requirements for the PARALLEL SPLITMERGE algorithm are of the order $O(N/K)$ for each processor. We now want to show that the MERGE procedure and its subordinate procedure, SPANNING, terminate, and that they correctly combine successive sibling clusters, while the resulting cluster sizes are smaller or equal to the page size.

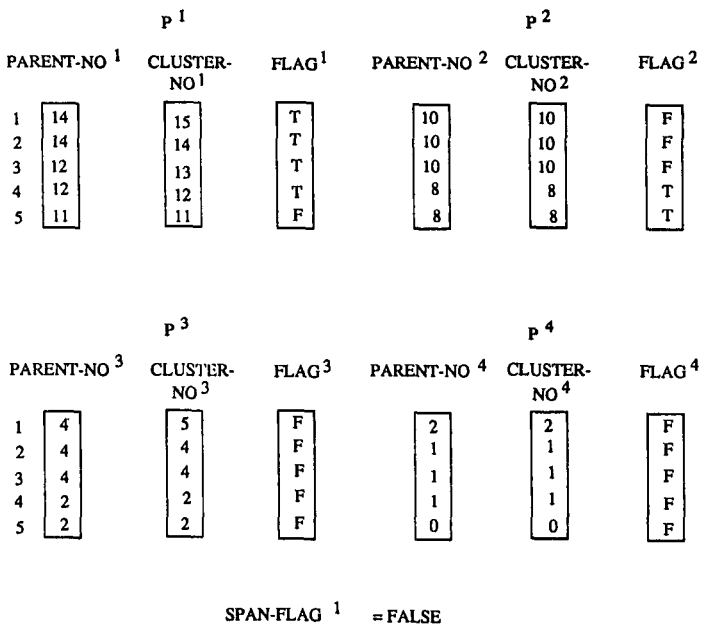


Fig. 9. Processor memory after SPANNING.

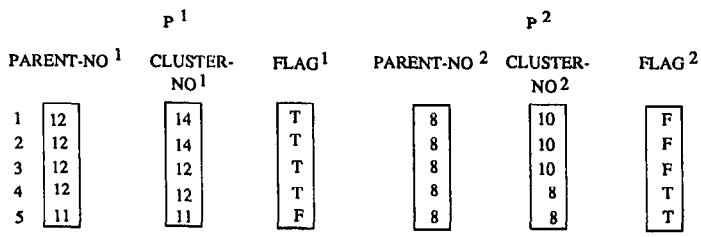


Fig. 10. Memory after Step 2.2 of second iteration of MERGE.

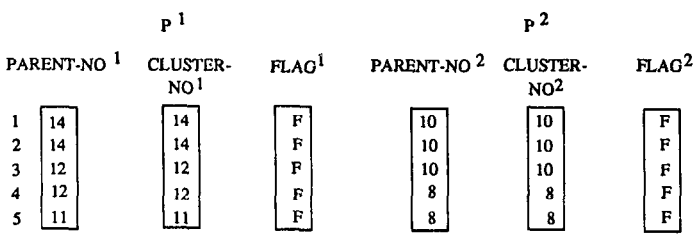


Fig. 11. Memory after Step 2.3 of second iteration of MERGE.

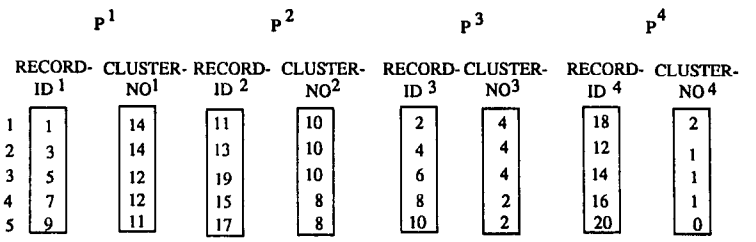


Fig. 12. Result of MERGE phase.

THEOREM. *Algorithms MERGE and SPANNING terminate correctly under the assumption that N/K is larger than, or equal to, the page size.*

PROOF. We first show that it suffices that processor P^j attempts to merge sibling clusters with the help of one adjacent processor only, P^{j+1} or P^{j-1}. A problem that could develop is for, say, processors P^j and P^{j+1} to fail to recognize a combined cluster that is larger than the page size, and, similarly, for P^{j+1} and P^{j+2} to fail in this task, but when taken together over P^j, P^{j+1}, and P^{j+2}, the cluster size does, indeed, exceed the page size. The combined cluster spans over three processors, and given our condition that $N/K \geq$ page size processor P^{j+1} would have set the FLAG^{j+1} vector to false in step 2.3 of MERGE; consequently, the FLAG^j and FLAG^{j+2} vectors would have been set to false in step 2.4 of MERGE. Thus, it is not necessary for a given processor to communicate to more than one processor, that is, P^{j+1}. A similar argument applies when P^j attempts to merge two different sibling clusters with P^{j-1}. Hence, a given processor, P^j, needs to communicate only with two adjacent processors, P^{j-1} and P^{j+1}. Next, we want to show that after a maximum of $M - 1$ iterations of MERGE, the FLAG^j vectors are set to false; hence the algorithm terminates. During a given iteration of MERGE, if two sibling clusters cannot be combined, their corresponding entries in FLAG^j (and maybe FLAG^{j+1}) are set to false in steps 2.3.2–2.3.4.

During the next iteration of MERGE, step 2.3.1 guards against the possible combination of nonsibling clusters. Consider, for example, two clusters C^1 and C^2 at level i in the P-tree which cannot be combined, and their "would-be" sibling WS at level $i - 1$ (that is, a cluster having the same grandparent in the P-tree). Due to the numbering scheme imposed on the P-tree, the next iteration of MERGE would find that the parent cluster numbers for WS coincide with those of either C^1 or C^2 and, consequently, also set the entries in the FLAG vector corresponding to WS to false. If the maximum number of iterations, $M - 1$, were executed, we would be left with two clusters, each spreading over the memory of at least one processor; hence all the FLAG^j must have been set to false. □

We estimate the time complexity of PARALLEL SPLITMERGE by looking at its three major components. In the SPLIT algorithm, each processor reads in parallel a subfile of size N/K and recomputes for its corresponding records their cluster numbers after each of the given M queries. Thus, the SPLIT phase requires $O(M * N/K)$ time. The parallel SORT algorithm, which we use in the second phase, has a time complexity of $O(N * \log N/K) + O(N)$, with the second

term accounting for the number of routing instructions. When $K = \log N$, we obtain for the SORT algorithm a time of $O(N)$. The while loop, that is, step 2, of the MERGE algorithm, is executed at most M times. During each iteration, a given processor can determine in time proportional to its local memory size, $O(N/K)$, whether the sibling clusters should be merged further, with the invocation of SPANNING also requiring at most $O(N/K)$ time. Thus, the MERGE and SPANNING phases have a time complexity of $O(M * N/K)$.

Overall, the time complexity of PARALLEL SPLITMERGE is $O(M * N/K + N)$. For $K \leq \log N \leq M$, the asymptotic speedup ratio of our algorithm, versus the original (serial) SPLITMERGE, is K, which is optimal.

5. EXPERIMENTAL RESULTS

In this section we report on a number of synthetic experiments we carried out to evaluate the performance of PARALLEL SPLITMERGE in terms of the quality of the clustering produced. The PARALLEL SPLITMERGE was implemented on a conventional architecture, i.e., using a single processor, since the objective was to estimate the average number of page accesses for the clustering produced. In [12] we reported on a number of experiments with our original SPLITMERGE algorithm. Our first objective was to compare the average number of page accesses obtained by SPLITMERGE versus PARALLEL SPLITMERGE for different data and query configurations. As expected the difference was negligible; in the worst case, PARALLEL SPLITMERGE was off 1.5 percent, a small price to pay for the algorithm speedup.

An additional set of experiments was performed to compare the performance of PARALLEL SPLITMERGE with the Adaptive Record Clustering algorithm of Yu et al. [23, 24], as well as with a randomly generated solution.

The Adaptive Record Clustering method of Yu et al. [23, 24] is a very general technique for adaptive database design [7]. Like our method, it works for arbitrary queries but does not require keeping statistics about query frequencies. However, the Yu et al. algorithm requires that a list of active record addresses, and their positions in the line $(-\infty, +\infty)$, be kept in random access memory. As each query occurrence is executed, the positions of the records satisfying the query are modified. First, the records are moved closer toward their centroid; in a subsequent step, they are moved apart from other records in order to permit identification of clusters. We describe below in more detail version 2 of the Yu et al. algorithm, which we denote by ARC2:

Step 0. *Initialization.* Each accessed record R_i is initially assigned an arbitrary position X_i on the line $(-\infty, +\infty)$.

Step 1. *Moving Records Together.* For each processed query Q_m, let $X_{m(i)}$, $1 \leq i \leq K$ be the positions of the records satisfying it. The centroid of the records accessed by Q_m is defined as:

$$CX = \sum_{i=1}^{K} X_{m(i)}/K. \tag{3}$$

The records in the query set are moved toward their centroid with a distance proportional to their currrent distance from it. $DX_{m(i)}$, the distance by which the

record with position $X_{m(i)}$ is moved, is defined as follows:

$$DX_{m(i)} = A * |X_{m(i)} - CX| * Benefit(Q_m)/BENMAX \quad (4)$$

where the benefit of query Q_i, Benefit(Q_m), is given by

Benefit(Q_m) = Cost2(Q_m) − Cost1(Q_m), and
Cost1(Q_m) = the minimum number of pages necessary to store the K records of the query set, and
Cost2(Q_m) = the expected number of page accesses necessary to retrieve the K records in the current configuration.

Similarly, the parameter, BENMAX, stands for the maximum benefit among all queries considered in a time interval, and A is a constant that we originally set to 0.5. As far as computing Cost2(Q_m), we assume that in the nonclustered configuration the K records are uniformly distributed among the S pages of the file, each containing N/S records, with N standing for the total number of records in the file.

Thus, we can make use of the formula developed by Yao [22], to obtain:

$$Cost\,2(Q_m) = S \times \left[1 - \prod_{I=1}^{K} \frac{N(1 - 1/S) - I + 1}{N - I + 1}\right]. \quad (5)$$

Step 2. Shift accessed records away. If the centroid of the records satisfying Q_m is less than the centroid of all accessed records, then each of the records in the query set of Q_m is shifted to the left by a distance equal to the average distance by which these K records were moved in Step 1. Otherwise, these records are shifted to the right by the same distance.

Step 3. Sort and form clusters. All the accessed records are sorted in ascending order of their positions. If the distance between two adjacent records is less than a distance, DK, defined as the average distance between adjacent records in the sorted list, then both records are assigned to the same cluster; otherwise a new cluster is identified.

Step 4. Assignment of records to pages. This step is similar to the physical phase of the SPLITMERGE algorithm. Our experiments were divided into a number of classes, from which we discuss below a sample of four:

CLASS	NO. OF RECORDS (N)	NO. OF QUERIES (M)	NO. RECORDS PER QUERY (MAX)
I	1000	100	100
II	1000	100	250
III	1000	50	100
IV	1000	50	50

The number of records specified in an occurrence of query type Q_m, denoted by $S(Q_m)$, is a random number in the range [1, MAX]. Once the cardinality of the query set has been determined, a number of $S(Q_m)$ record identifiers, drawn from the range [1, N], are generated for a particular query occurrence.

The query type frequencies were also synthetically produced and varied from a highly skewed distribution to one in which the frequencies were the same for

Table I. Query Distributions

Distributions	Queries	Frequencies
1	1–5	0.1042
	6–20	0.0208
	21–100	0.0021
2	1–10	0.0309
	11–30	0.0206
	31–60	0.0052
	61–100	0.0031
3	1–10	0.018
	11–60	0.01
	61–100	0.008
4	1–60	0.0105
	61–90	0.0095
	91–100	0.0084
5	1–100	0.01
6	1–25	0.0222
	26–50	0.0111
	51–75	0.0056
	76–100	0.0011
7	1–5	0.0205
	6–30	0.0154
	31–35	0.0144
	36–70	0.0082
	71–100	0.0051
8	1–5	0.05
	6–25	0.02
	26–75	0.005
	76–100	0.004
9	1–1	0.1003
	2–51	0.015
	52–100	0.003
10	1–5	0.08
	6–30	0.01
	31–100	0.005

all queries. For each class of experiments, we used the ten query distributions exhibited in Table I. For the PARALLEL SPLITMERGE algorithm, we ran 1000 query occurrences for each experiment, given class and query distribution. With regard to the ARC2 algorithm, an additional concern to be considered was the convergence of the method [24]. We selected time intervals of 400 query occurrences, drawn from the same distributions in Table I, and after each interval we computed the resulting allocation to physical storage. If the difference between two consecutive allocations (mappings) to storage was small [24], the algorithm terminated; otherwise, we repeated the steps for a new time interval.

The experimental results for Class 1 are depicted in Table II. The table contains the average number of page accesses per query for a given clustering approach, as well as the percent difference between the corresponding pairs of methods. The average number of page accesses necessary for a random solution (abbreviated as RN) was obtained using Yao's formula [22] in step 1 of the ARC2 method.

Table II. Comparing Parallel and ARC2 Methods for Class I (Average Pages/Query)

	PARALLEL	ARC2	RANDOM	ARC2/PARALLEL DIFF	PARALLEL/RN DIFF	ARC2/RN DIFF
1	13.71	24.81	39.53	80.96	65.32	37.24
2	19.17	31.72	35.91	65.47	46.62	11.67
3	25.74	32.04	35.51	24.48	27.51	9.77
4	27.28	32.02	35.24	17.38	22.59	9.14
5	27.53	32.25	35.56	17.14	22.58	9.31
6	19.08	30.54	33.55	60.06	43.13	8.97
7	23.46	31.38	34.80	33.76	32.59	9.83
8	19.15	30.76	36.73	60.63	47.86	16.25
9	18.76	28.36	33.43	51.17	43.88	15.17
10	18.80	27.62	37.89	46.91	50.38	27.10

Average % difference between ARC2/PARALLEL: +45.80%
Average % difference between PARALLEL/RN: −40.25%
Average % difference between ARC2/RN: −15.45%
Average no. of pages per query for PARALLEL: 21.27
Average no. of pages per query for ARC2: 30.15

In Table II, we see that the PARALLEL SPLITMERGE algorithm outperforms ACS2 in all trials. On the average, PARALLEL SPLITMERGE produces a solution that uses 40 percent fewer page accesses than the random solution, while ARC2 produces a solution that uses 15 percent fewer page accesses. We noticed that with the ARC2 method records tend to bunch up in a few clusters. A secondary reason why ARC2 may not yield better results is due to its treatment of infrequent queries. If the occurrences of infrequent query type are processed, then this would cause ARC2 to pull certain records together. However, if these records were also required earlier by other more frequent queries, then these records would be pulled apart from the centroids of the more frequent queries. Thus, records belonging to a number of queries (that is, the secondary cluster of a primary cluster) are likely to end up in positions quite apart on the line, which increases the number of page accesses.

For the Class 2 experiments, represented in Table III, the maximum number of records that can be requested by a query is increased from 100 to 250, while the size of the database does not change. The decrease in performance is caused by a greater overlap between the query occurrences. ACS2 and PARALLEL SPLITMERGE improve the random solution on the average by only 7 and 24 percent respectively. Nevertheless, a 24 percent improvement is appreciable.

In Table IV (Class 3), we decrease the number of queries to 50, with a maximum of 100 records per query and, in Table V (Class 4), we decrease further the maximum number of records per query to 50. This has the effect of decreasing the overlap among queries, in comparison to Class 1, and we can see a marked improvement in both ARC2 and PARALLEL SPLITMERGE. The random solution is improved by 53 percent for Class 3, by 67 percent for Class 4 by PARALLEL SPLITMERGE, and by 28 and 48 percent, respectively, by ARC2. One additional observation about ARC2 for Classes 3 and 4 is that, instead of the records congregating in a few clusters, most of the records formed individual clusters.

Table III. Comparing Parallel and ARC2 Methods for Class II (Average Pages/Query)

	PARALLEL	ARC2	RANDOM	ARC2/PARALLEL DIFF	PARALLEL/RN DIFF	ARC2/RN DIFF
1	30.86	51.97	69.84	68.41	55.81	25.59
2	47.26	58.62	62.99	23.25	24.97	6.94
3	53.99	61.19	61.98	13.34	12.89	1.27
4	56.59	57.62	61.77	1.82	8.39	6.72
5	56.19	60.46	61.56	7.53	8.72	1.79
6	50.23	61.53	63.99	22.50	21.51	3.84
7	54.06	62.87	63.69	11.30	15.12	1.29
8	43.73	62.57	66.24	43.08	33.98	5.54
9	48.80	56.36	61.89	15.49	21.14	8.94
10	40.44	60.41	68.30	49.38	40.79	11.55

Average % difference between ARC2/PARALLEL: +25.61%
Average % difference between PARALLEL/RN: −24.33%
Average % difference between ARC2/RN: −7.35%
Average no. of pages per query for PARALLEL: 48.21
Average no. of pages per query for ARC2: 59.36

Table IV. Comparing Parallel and ARC2 Methods for Class III (Average Pages/Query)

	PARALLEL	ARC2	RANDOM	ARC2/PARALLEL DIFF	PARALLEL/RN DIFF	ARC2/RN DIFF
1	9.78	14.69	34.57	50.20	71.71	57.51
2	14.71	25.80	36.58	75.39	59.77	29.47
3	22.74	33.12	38.54	45.65	41.00	14.06
4	23.80	33.88	39.35	42.35	39.52	13.90
5	23.64	32.82	39.71	38.83	40.46	17.35
6	15.80	27.82	37.43	76.08	57.78	25.67
7	19.81	28.35	37.59	43.11	47.30	24.58
8	15.10	26.13	37.22	73.05	59.43	29.80
9	14.65	24.74	33.16	68.87	55.81	25.39
10	14.56	20.59	35.41	41.41	58.88	41.85

Average % difference between ARC2/PARALLEL: +55.49%
Average % difference between PARALLEL/RN: −53.17%
Average % difference between ARC2/RN: −27.96%
Average no. of pages per query for PARALLEL: 17.46
Average no. of pages per query for ARC2: 26.79

It is now appropriate to point out some of the subtle differences between PARALLEL SPLITMERGE and ARC2. Our algorithm requires that the query statistics be collected and classified before the algorithm can be run, while ARC2 is an adaptive algorithm that computes the record positions and shifts them as part of the retrieval process. Our method, then, requires that we identify syntactically identical queries that retrieve the same set of records, and we are currently investigating this problem. The time complexity of our method is $O(M * N/K + N)$, where M is the number of the most frequently incurred queries. Since no information about query frequencies is kept in the ARC2 algorithm, its steps must be repeated for all queries (i.e., the additional cost to retrieval due to running the clustering steps is $O(M' * N)$, where M' is the total

Table V. Comparing Parallel and ARC2 Methods for Class IV (Average Pages/Query)

	PARALLEL	ARC2	RANDOM	ARC2/PARALLEL DIFF	PARALLEL/RN DIFF	ARC2/RN DIFF
1	5.54	7.83	26.34	41.34	78.96	70.27
2	7.01	12.50	23.33	78.32	69.97	46.42
3	8.31	12.97	21.01	56.08	60.43	38.27
4	8.53	10.98	20.17	28.72	57.72	45.56
5	8.62	12.08	20.01	40.14	56.92	39.63
6	7.51	13.19	23.06	75.63	67.43	42.80
7	7.83	12.69	21.40	62.07	63.41	40.70
8	6.81	11.08	23.50	62.70	71.02	52.85
9	7.01	11.65	22.81	66.19	69.27	48.93
10	6.54	9.72	23.89	50.15	72.62	59.31

Average % difference between ARC2/PARALLEL:	+56.13%
Average % difference between PARALLEL/RN:	−66.78%
Average % difference between ARC2/RN:	−48.47%
Average no. of pages per query for PARALLEL:	7.37
Average no. of pages per query for ARC2:	11.47

number of queries). In our method, the cost in page accesses of reading the input file in parallel is $O(N/K * \text{PAGESIZE})$. Although the ARC2 method makes the assumption that the record positions are all available in random access memory, this may not be feasible for large databases; hence, additional page accesses may occur here too.

In summary, we have seen that in all tests our algorithm outperforms ARC2, the only other method dealing with arbitrary queries. Our method produces substantial savings when applied to an environment with a nonuniform distribution of query frequencies and/or with relatively little overlap among the queries.

6. CONCLUSION

We have presented an efficient parallel algorithm for record clustering that can run on a SIMD machine whose processors are connected via an interconnection network. We have introduced the P-tree and its numbering scheme, which allows each processor to perform the initial allocation of records to clusters; we have shown that by restricting the merging to sibling clusters, we can reduce the amount of interprocessor communication and that the difference in performance, versus the original SPLITMERGE algorithm, is negligible.

Our algorithm can be applied to an environment consisting of arbitrary queries whose frequencies of request and selectivities can be estimated. In practice, it suffices to restrict ourselves to the most frequently appearing queries only, which should eliminate a large proportion of them. Our experimental results have shown that our method produces substantial savings when applied to an environment with a nonuniform distribution of query frequencies and with relatively little overlap among the queries.

Important questions to be addressed, in the general context of database design, are how much information and which statistics about the query structure should be kept and how should this information be stored. We envision that our method

could be easily applicable to object-oriented databases, for which the permissible query structure is known in advance and stored by the system as part of the object description. However, even for more traditional database systems, the statistics about query frequencies are needed in order to perform different optimization tasks, such as index selection. As we have shown, our algorithm outperforms ARC2, the only other record-clustering method that deals with arbitrary queries, and lends itself to easy parallelization.

We view record clustering not as a one-time operation, but as an operation that may need to be executed periodically, if the query structure changes substantially. In using the adaptive clustering approach of Yu et al. [24], the algorithm needs to be run continuously, in order to detect whether a reorganization is warranted. We are currently investigating how the additional information available in our approach (that is, query frequencies and query set sizes) can be used to determine the points in time when a change in the query structure warrants a new clustering and subsequent reorganization of the file.

ACKNOWLEDGMENT

We would like to thank Dina Bitton for her insightful comments on an earlier version of this paper.

REFERENCES

1. AHO, A., HOPCROFT, J., AND ULLMAN, J. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass., 1983.
2. BANERJEE, J., BAUM, R. I., AND HSIAO, D. K. Concepts and capabilities of a database computer. *ACM Trans. Database Syst. 3*, 4 (1978), 347–384.
3. BAUDET, G., AND STEVENSON, D. Optimal sorting algorithms for parallel computers. *IEEE Trans. Comput. C-27*, 1 (1978), 84–87.
4. BITTON, D., DEWITT, D., HSIAO, D., AND MENNON, J. A taxonomy of parallel sorting. *ACM Comput. Surv. 16*, 3 (1984), 287–318.
5. BORAL, H., DEWITT, D., BITTON, D., AND WILKINSON, K. Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst. 8*, 3 (1983), 324–353.
6. DU, H. C., AND SOBOLEWSKI, J. S. Disk allocation for Cartesian product files on multiple-disk systems. *ACM Trans. Database Syst. 7*, 1 (1982), 82–101.
7. HAMMER, M., AND NIAMIR, B. A heuristic approach to attribute partitioning in a self-adaptive database management system. In *Proceedings of the International Conference on Management of Data* (Washington, D.C., 1976). 1–8.
8. HIRSCHBERG, D. S. Fast parallel sorting algorithms. *Commun. ACM 21*, 8 (Aug. 1978), 657–661.
9. JAKOBSSON, M. Reducing block accesses in inverted files by partial clustering. *Inf. Syst. 5*, 1 (1980), 1–5.
10. LIOU, J. H., AND YAO, S. B. Multidimensional clustering for database organization. *Inf. Syst. 2*, 4 (1977), 187–198.
11. NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. C. The grid file: An adaptable, symmetric multiway file structure. *ACM Trans. Database Syst. 9*, 1 (1984), 38–71.
12. OMIECINSKI, E. Algorithms for record clustering and file reorganization. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Northwestern Univ., Evanston, Ill., 1984.
13. OMIECINSKI, E. Incremental file reorganization schemes. In *Proceedings of the 11th International Conference on Very Large Databases* (Stockholm, 1985), 346–357.
14. OMIECINSKI, E., AND SCHEUERMANN, P. A global approach to record clustering and file reorganization. In *Proceedings of the 3rd Joint BCS/ACM Symposium on Research and Development in Information Retrieval*, C. J. van Rijsbergen, Ed. Cambridge University Press, 1984, 210–221.

15. ORENSTEIN, J. A., MERRETT, T. H., AND DEVROYE, L. Linear sorting with $O(\log n)$ processors. *BIT* 23 (1983), 170–180.
16. OUKSEL, M., AND SCHEUERMANN, P. Storage mappings for multidimensional linear dynamic hashing. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., 1983). ACM, New York, 1983, pp. 90–105.
17. QUINN, M., AND DEO, N. Parallel graph algorithms. *ACM Comput. Surv. 16*, 3 (1984), 319–348.
18. SAVAGE, C., AND JAJA, J. Fast, efficient parallel algorithms for some graph problems. *SIAM J. Comput. 10*, 4 (1981), 682–691.
19. SCHEUERMANN, P., AND OUSKEL, M. Multidimensional B-trees for associative searching in database systems. *Inf. Syst. 7*, 2 (1982), 123–137.
20. TEOREY, T., AND FRY, J. *Design of Database Structures.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
21. VALDURIEZ, P., AND GARDARIN, G. Multidimensional join algorithms of relations. In *Improving Database Usability and Responsiveness*, P. Scheuermann, Ed. Academic Press, New York, 1982, pp. 237–256.
22. YAO, S. B. Approximating block accesses in database organizations. *Commun. ACM 20*, 4 (April 1977), 260–261.
23. YU, C. T., SIU, M. K., LAM, K., AND TAI, F. Adaptive clustering schemes: General framework. In *Proceedings of IEEE COMPSAC Conference* (Chicago, 1981). 81–90.
24. YU, C. T., SUEN, C. M., LAM, K., AND SIU, M. K. Adaptive record clustering. *ACM Trans. Database Syst. 10*, 2 (1985), 180–205.

Received September 1986; revised September 1988; accepted October 1989