A FRAMEWORK FOR DISTRIBUTING AND PARALLELIZING SECONDARY MEMORY SEARCH STRUCTURES

(Extended Abstract)

by

G. Matsliach and O. Shmueli

Technical Report #637
June 1990
A Framework for Distributing and Parallelizing Secondary Memory Search Structures
(Extended Abstract)

Gabriel Matsliach Oded Shmueli
Technion – Israel Institute of Technology
Dept. of Computer Science

1 Introduction

Traditional architectures which consist of a single processor, a small main memory, and secondary memory devices (e.g. disks), are gradually being replaced by multiprocessor multi-disk architectures with a relatively large main memory. The processors either reside in the same site and communicate via the shared memory, or reside in different sites and communicate via a local network. We consider architectures which consist of a relatively small number of processors $P$ (say 2-20), and each processor possesses a disk (or a group of disks).

A fundamental issue when designing file systems in such architectures is the design of search structures for maintaining single key files, multi-key files, and indices. The search structures may be distributed among the processor-disk pairs in order to achieve parallel execution capabilities, and an effort should be made to reduce the consumption of resources such as main memory.

The design and implementation of traditional search structures are well understood. Examples of such traditional search structures are $B^+$-trees and their variants [C], dense indices [U], Extendible Hash tables [FNPS], Bounded Disordered Files [L02], $T^*$-trees [L], k-d trees [B], Grid Files [NIS], K-D-B-trees [R], hB-trees [LS] etc. Therefore, the standard approach is to keep using the traditional search structures within the modern architecture, while making an effort towards a distribution of the structure's components among the processors and disks, in order to achieve parallel execution capabilities [LB]. A similar approach [DGGHKM] [MS1] is to equally partition the file records among processors (for example, by using a hash function), each of which maintains its part in a local traditional search structure, stored in its disk. We argue that additional benefits (besides parallel capabilities) can be achieved by modifying the traditional search structures, in order to achieve a better fit to multiprocessor multi-disk environments.

In this paper, we propose a framework for distributing search structures in multiprocessor multi-disk environments. Given a traditional search structure, its distribution process, according to our framework, consists of three parts:

1. **Structural design**: The components of the search structure\(^1\) are modified by changing their structure (various components may be modified differently).

2. **Mapping function**: A storage device is determined for each of the search structure's components. An example of a mapping of a component is as follows:

\(^1\)An example of a search structure component is a node in a search tree.
"the component is equally partitioned into \( P + 1 \) parts, each of the first \( P \) parts is stored in a different processor's disk, and the last part is stored in main memory."

3. **Gine**: the structural design is combined with the mapping function by specifying coordination and synchronization among processors, and designing the algorithms for executing the basic operations (e.g. search, insert, delete, and range operations) on the resulting distributed structure, and guaranteeing a consistent parallel activity.

The framework defines a family of structural operators. The structural modification of a search structure's component is determined by deciding which combination of structural operators is to be applied to the component. The whole structural design is defined by determining the structural modification of each of the components of the given (non-distributed) search structure.

Each composition of a structural design and a mapping function defines a distribution method for traditional search structures in multiprocessor multi-disk environments. Many distribution methods that were previously designed \([\text{MS1} \text{ MS2} \text{ MS3} \text{ MS4}]\), can be defined using the framework. Defining a framework provides a simple mechanism for designing and generating new methods for distributing search structures. The evaluation of the resulting distribution methods is done according to the following criteria:

1. The degree of parallel capabilities.
2. The savings in the average execution time per operation and the average sojourn time per operation.\(^2\)
3. The saving in main memory space consumption (when parts of the search structure need to be stored in main memory).
4. The average disk space utilization.
5. The degree of operational dependency among processors and the cost of using the network (when the processors are in different sites).

The framework presented in this paper is by no means exhaustive. It can be augmented with additional structural operators, additional groups of operators, and additional modes of enlargement (see below). It does capture, however, a very large family of distribution methods. (An analogous approach with respect to distributed concurrency control algorithms was taken by Bernstein and Goodman in \([\text{BG}]\)).

### 2 The framework

Most relational database systems maintain secondary memory data (i.e. files and indices) by using search structures that can be thought of as directed acyclic graphs with a single root. Therefore, we consider these types of search structures. Note that most of the commonly used search structures (including all the structures mentioned in the introduction) are of this type. Thus the terms root, leaves, and internal nodes, are well defined. Each basic unit of a search structure is called a component of the search structure. In these structures, an overflowed (or a heavily loaded) component is enlarged by either increasing its size (e.g. a partial expansion \([\text{Lo1}]\)) or by creating a new component into which some of the data (of the full component) is moved (e.g. a split).\(^3\) Additional modes of enlargement could, in principle, be added to the framework.

\(^2\)The sojourn time is the sum of the waiting time before execution and the execution time.

\(^3\)Note that there are structures (such as the \(B^*\) tree \([\text{SF}]\)) in which some of the siblings of the overflowed component may be involved in the splitting as well.
We now detail the framework for distributing search structures in multiprocessor multi-disk environments. In the next section, we give three examples of distribution methods, and show how they can be defined using this framework.

### 2.1 First step: the structural design

Given a search structure, the structural modification is defined for each of its components by choosing the structural operators to be applied to the component. The structural operators proposed in the framework are partitioned into five groups, and the structural modification of a component is defined by choosing one operator from each group and composing the chosen operators in group order (1 through 5). The second group contains a “no change” operator, denoted by NOP. If NOP is chosen, then the structural modification is complete (and the remaining three groups are ignored). We now detail the operators in each of the five groups:

**Group 1: changing the component size**

1.1 **EXPAND(S)** - The overall space allocated to the component is multiplied by S. In other words, the component size is modified by a factor of S.

   If the components of the original non-distributed search structure are logically partitioned into sub-components (for example, multi-bucket components), then the exact manner of the size modification need be specified. Therefore, when such components are considered, one of the following three operators will be chosen (instead of operator 1.1):

   1.2 **BLOW(S)** - The component size is modified by a factor of S by modifying each of its sub-components by a factor of S (the number of sub-components remains unchanged). For example, applying this operator to a multi-bucket component means that the size of each bucket is modified by a factor of S (and the number of buckets is not changed).

   1.3 **MULTIPLY(S)** - The component size is modified by a factor of S by multiplying it S times (S is a natural number). For example, applying this operator to a multi-bucket component which consists of L buckets means that the number of buckets is multiplied by S while their size is not changed (the modified component thus consists of S*L buckets).

   1.4 **MULT-BLOW(Q,R)** - The component is blown by a factor of R and then multiplied Q times. The total size modification is thus S=Q*R.

**Remark:** Usually, a system has one or a few basic disk allocation units, namely, it can allocate contiguous disk space in a few basic sizes (in terms of number of disk pages). Therefore, the choice of S in operators 1.1 and 1.2 (and the choice of R in operator 1.4) may result in a component whose size is not equal to that of an integral number of basic disk allocation units. This may lead to disk space waste. (Such a wasted space may be caused even if the system can allocate any number of contiguous disk pages.) Thus, one should choose S carefully based on system characteristics. Observe that if a component size in the non-distributed structure is r pages, then probably the system can allocate units of r contiguous disk pages. Thus, if S is a natural number then a wasted space can be prevented.

**Group 2: a logical partition of the component into autonomous cuts**

The following operators define several possibilities for an internal partition of a component (following its size modification) into contiguous portions (of the component) called cuts. While a component that was not originally partitioned (i.e., a component for which operator 1.1 was
chosen) can be partitioned, in principle, in any desired manner, a partition into cuts, of a component that was originally partitioned (into sub-components), must not cut within sub-components. Thus, any desired partition (into cuts) of an originally partitioned component is possible provided that each cut contains an integral number of sub-components (blown sub-components, in the case of operator 1.2 or 1.4 being applied to the component). The operators are:

2.1 NOP - The component is not partitioned any further and is viewed as a single logical unit.
2.2 DIVIDE(X) - The component is divided into X autonomous cuts of equal size, each of which is a contiguous portion of the component. In the case of an originally partitioned component, each cut must contain an integral number of sub-components (blown sub-components, in the case of operator 1.2 or 1.4 being applied to the component).
2.3 PARTITION(X₁ ... Xₘ) - Let \( W = \sum_{i=1}^{m} X_i \). The component is divided into \( m \) autonomous cuts (each of which is a contiguous portion of the component), where the size of the \( i \)-th cut is \( X_i/W \) of the component. In the case of an originally partitioned component, each cut must contain an integral number of sub-components (blown sub-components, in the case of operator 1.2 or 1.4 being applied to the component).

Group 3: handling overflows from cuts

The operators of this group deal with overflow at the cut level. The possibility of an internal cut overflow area is not precluded.

3.1 NO-OVF - A cut that overflows must be enlarged.
3.2 OVF(j) - The \( j \)-th cut is designated as an overflow cut. In other words, data items that overflow are stored in the \( j \)-th cut. Once the \( j \)-th cut overflows, the whole distributed component need be enlarged.
3.3 SECOND-TRY - When a data item with a key \( k \), which is directed to cut \( i \), overflows, an effort is made to store it in another cut denoted by \( st(i,k) \). If the \( st(i,k) \)-th cut is full, then the \( i \)-th cut need be enlarged. This operator is, in fact, a family of operators as we have not specified how \( st(i,k) \) is determined. One possibility is to determine \( st(i,k) \) by using a hash function. We note that this family of operators induces a run-time bookkeeping overhead.

Group 4: dependency among cuts with respect to enlargements

As explained, each cut is regarded logically as an autonomous portion of the component. Thus, when a cut overflows (or becomes heavily loaded) it need be enlarged. The following operators define several approaches to the dependency among the cuts of a component, with respect to enlargements. Note that by an enlargement of the entire component we mean an enlargement of each of its cuts.

4.1 FULL-DEPENDENCY (FULL-DEP) - Once a cut need be enlarged (via split or partial expansion), all cuts are enlarged. There is thus a full dependency among the cuts.
4.2 SPLIT-DEP-EXP-INDEP - This operator is introduced for search structures that perform partial expansions before splitting [Lot]. The operator means that partial expansions of cuts are performed independently (of the other cuts), but once a cut need be split, all cuts are split. Thus, dependency among cuts exists only with respect to splits.
4.3 SPLIT-PART-DEP - This operator is introduced for enlargements which are implemented as splits. Once a cut \( u \) need be split, only \( u \) is split while the other cuts are not. Once a cut need be split again (i.e., it was created by an independent split of some cut of this component), then all cuts that have not yet been split, are split. Thus, only partial dependency exists among
the cuts with respect to splits. (An example of such an approach can be found in [MS4]). For search structures that perform partial expansions, this operator means that partial expansions are performed independently (see operator 4.2) and in addition, only a partial dependency exists between the cuts with respect to splits (as explained above).

4.4 FIXED – The cuts are independent. The influence of a cut enlargement does not affect the other cuts. For example, consider a collection of S independent B+-trees. This collection of S trees is isomorphic to a tree, called the “collection tree”, whose root is a collection of S independent cuts, where the i-th cut is the root of the i-th tree of the collection of trees. Once the i-th cut (i.e. the root of the i-th tree) overflows, it is replaced with the new root of the i-th tree (and thus a new level is created in the i-th tree, namely, in the subtree whose root is the i-th cut of the “collection tree”). The root of the “collection tree” is an example for a FIXED partitioned component.

Group 5: partitioning the data among the cuts

5.1 HASH - The cut for a newly inserted item is determined by using a hash function.

5.2 RANGE - The range of the possible key values in the component is partitioned among the cuts (this partition of the key ranges is predefined, but not necessarily uniform).

The structural modification of a component is described by composing the structural operators which are applied to it in group order (the operators are separated by the symbol “;” which denotes function composition). Recall that the structural design is defined by defining the structural modification for each structure component. Examples are given in section 3 where we describe three distribution methods and show how they can be defined using the framework.

2.2 Second step: the mapping function

The mapping function determines, once the structural design is complete, how each of the structure's components is distributed among the disks and main memory. The mapping of a component is not necessarily at the cut level; the mapping function may divide a cut into a few parts and map the different parts to various disks. In order to prevent disk space waste, it is recommended that the size of each resulting part which is mapped to the disk be an integral number of basic disk allocation units.

If the mapping function divides a component (cut) into equal parts and maps each part to a different disk, then it may require that the parts be stored in the same physical address in the different disks (unless the system can not accommodate such a requirement). If all the component's parts are stored in the same physical address (in the different disks), then a pointer that points at the component need consist of a single physical disk address. However, if the different parts are stored in different addresses (for example, because two parts are mapped to the same disk), then a pointer to that component consists of several physical disk addresses.

2.3 The glue

Once the structural design and the mapping function are determined, they must be “glued” in order to become a functioning distribution method. This process is not always straightforward. Three aspects that need be considered are:
The distributed structure's pointers: Conceptually, it is possible that a pointer need contain information concerning the "type" of the component to which it points, because the different structure's components may have a different structure (due to a different structural modification) and may be differently distributed among disks. For example, if all the leaves have the same type, and all internal nodes are of the same type, then each pointer need contain one bit which indicates whether the component to which it points is an internal node or a leaf. In general, more information might need to be kept in some of the distributed structure's pointers.

Coordination among processors: As explained, the distributed search structure's components may be distributed among a number of processor-disk pairs. Therefore, an access to a component may require processors to access their disks independently (to read the component's parts). Thus, the coordination and synchronization of the processors is an essential issue which must be considered in order to guarantee a correct and efficient distributed structure.

Concurrent accesses to components: There are distribution methods (which are generated by the framework) in which different search paths (of concurrently executed operations) may concurrently access different cuts of the same component which are stored in different disks. As an enlargement of a component (for example, a split) affects all its cuts, inconsistency may be caused by two such updates (which are concurrently executed) whose search paths are not disjoint (with respect to components). In such distribution methods, a consistent parallel activity must be guaranteed (for example, by composing the basic algorithms with algorithms for concurrent search structures [BG] [S]). It is thus crucial, when designing a distribution method, to check if the resulting distributed search structures require mechanisms that guarantee a consistent parallel activity.

3 Examples of distribution methods

In this section we provide examples of distribution methods and their definitions within the framework. We concentrate on the structural design and mapping parts, and reference the glue component to relevant publications.

3.1 SFM

In the introduction, we have noted that the standard approach for maintaining files and indices in multiprocessor multi-disk environments, is to keep using the traditional search structures, but to distribute their components (as a basic distribution unit) among the processors. In the terminology of the framework presented here, no structural modifications are made, and the mapping function maps the components, as a whole, to the disks. The main benefit of such an approach is achieving parallel execution capabilities (of operations that access different disks) and thereby decreasing the waiting time per operation before execution.

All the benefits that can be obtained using the above standard approach, seem to be realized by the following straightforward distribution method (denoted by SFM). Given a traditional search structure of some type X that maintains a file, it is distributed by SFM as follows: the file records are distributed among the P processors using a hash function, and each processor maintains its part of the file in a "local" search structure of type X which is stored in the processor's disk. (If some parts of the original search structure are stored in main memory, then
the same parts of each local structure are stored in main memory). The definition of SFM using the framework is as follows:

**Structural design:** all the structure's components, except for the root, are not changed. Thus, their "structural modification" is: NOP o EXPAND(1). The root is modified to capture the idea of a collection of \( P \) independent trees (see operator 4.4), by the following structural modification:

\[
\text{HASH} \circ \text{FIXED} \circ \text{NO} \circ \text{OVF} \circ \text{DIVIDE}(P) \circ \text{EXPAND}(P) \left[ \text{root} \right]
\]

**Mapping function:** the \( i \)-th cut of the root is mapped to processor \( i \)'s disk. Each component whose search path passes through the \( i \)-th cut of the root (\( 1 \leq i \leq P \)) is mapped to processor \( i \)'s disk.

New distribution methods are evaluated by comparing them to SFM (under the criteria mentioned in the introduction). This is because SFM is essentially at least as good as all the distribution methods that apply the standard approach, namely, do not make any structural modifications. In the next two subsections, we briefly describe two distribution methods [MS2] [MS4], show how they are defined using the framework, and explain their advantages over SFM. We are currently designing and evaluating additional distribution methods, based on the framework. For example, the use of the operator RANGE (instead of HASH), and the use of an overflow cut.

### 3.2 A general distribution method that saves main memory space

We present a distribution method for a large family of traditional search structures (which contains all the structures which were mentioned in the introduction). This family contains search structures that are composed of an index (or a directory) and a collection of data leaves. The index is stored in main memory and its entries point at the data leaves which are stored in the disk and contain the data records. (If the index does not fit into the main memory space allocated for the file, then some of the index components are stored in the disk.) When distributing these search structures using SFM, each local structure is composed of an index (stored in a processor's main memory) that points to the data leaves (which are stored in the processor's disk), whose number is roughly \( 1/P \) of the number of data leaves in the original non-distributed structure. As an index size is essentially proportional to the number of leaves pointed by it, the total main memory space consumption of search structures distributed by SFM is at least as large as the main memory space consumption of the original non-distributed structure.

The distribution method proposed here generates distributed search structures which have low main memory space requirements (the exact reduction in main memory space consumption with respect to SFM is 50-70%), the same execution cost per operation (w.r.t. SFM), the same degree of parallelism, and only a marginal reduction in disk space utilization. Thus, a significant saving in main memory space is obtained, with almost no negative effects (this was verified by analyses and simulations).

The basic idea is the use of large data leaves which are \( P \) times the size of a leaf in the original non-distributed structure. These leaves are equally partitioned into \( P \) cuts, and each cut is stored in a different processor's disk. A cut of a leaf is chosen for a record using a hash function. A single index, which points at the large leaves, is maintained. This explains the reduction in main memory space consumption. The cuts of such a large leaf are only partially dependent (see [MS4] for details). The definition of this distribution method using the framework is as follows:

**Structural design:** all the structure's components, except for the leaves, are not changed. Thus, their "structural modification" is: NOP o EXPAND(1). The structural modification of
the leaves is as follows:

\[ \text{Mapping function: all the components, except for the leaves, are stored in main memory. The } \]
\[ \text{i-th cut of each leaf is stored in processor i's disk (} 1 \leq i \leq P\text{). The different cuts of a leaf are not necessary stored at the same physical address (in the different disks) and thus an index pointer that points at a leaf consists of } P \text{ physical addresses.} \]

Remark: when search structures that have multi-bucket leaves are considered, the first operator which is applied to the leaves is \( \text{MULTIPLY}(P) \).

### 3.3 Distributing secondary memory B+-trees

When only a small amount of main memory space is allocated to a file (either because the file is not accessed frequently or because the system's main memory is small), the appropriate traditional search structure for maintaining the file is the B+-tree. We have designed the following distribution method for B+-trees [MS2]. The \( P \) processor-disk pairs are partitioned into several disjoint groups. A group which consists of \( D \) processor-disk pairs (\( D \leq P \)) implements a wide B+-tree whose nodes are \( D \) times larger (than the original non-distributed structure). The resulting degree of this B+-tree is \( D \) times larger than that of the original non-distributed B+-tree, and it is thus shorter (usually by one level). Each node in this wide B+-tree is physically partitioned among the disks of the group (which implements the tree) and thus, in order to access the node all processors of the group simultaneously access their disks. Each processor can determine whether the search should proceed through its part of the node without communicating with the other processors (see [MS1] for details). The trees implemented by the groups are independent (similar to SFM), and a tree (i.e. a group) is selected for a record by using a hash function.

The execution time per operation is smaller in this method (w.r.t. SFM) as the trees are shorter, but the degree of parallelism is equal to the number of groups, rather than to the number of processors \( P \) (as in SFM). Thus, the reduction in execution time per operation is obtained at the expense of parallel execution capabilities. Approximate analyses and extensive simulations showed that the average sojourn time per operation in this method is reduced by about 30\% as compared to the respective time for SFM. The definition of this distribution method using the framework is as follows:

**Structural design:** Given \( P \) processor-disk pairs, let \( T \) be the smallest number such that the height of a local B+-tree of SFM (which maintains \( 1/P \) of the file's records) is larger by one than the height of a B+-tree with a degree larger by a factor of \( T \), which maintains \( T/P \) of the file's records (assume here, for simplicity, that \( P \mod T = 0 \)). The structural modification of all the structure's components, except for the root, is: \( \text{NOP} \circ \text{EXPAND}(T) \). The root is modified to capture the idea of a collection of \( P/T \) independent trees. Thus, the structural modification of the root is:

\[ \text{Mapping function: The i-th cut of the root is equally partitioned among the } T \text{ disks of the i-th group (for all } 1 \leq i \leq P/T \text{) by "cutting" it in } T-1 \text{ points. Each node (which is not the root) whose search path passes through the i-th cut of the root (} 1 \leq i \leq P/T \text{), is equally partitioned among the } T \text{ disks of the i-th group by cutting it in } T-1 \text{ points. The different parts of each node are stored in the same physical address in the disks of the group maintaining the node.} \]
References


