# Microprocessor Assisted Tuple Access, Decompression and Assembly for Statistical Database Systems[1]

Paula Hawthorn

Lawrence Berkeley Laboratory

**Abstract**

A methodology is formulated for determining if a microprocessor-based specialized system is practical for the solution of a problem in an application. This methodology is followed in the description and justification of of a back-end system for use with a statistical database system. The functions to be performed by the back-end system include those related to attribute partitioning, compression, and data access. It is shown that the system designed (the Microprocessor Assist System) will be cost-effective in increasing the performance of a statistical data management system.

Additionally, a category-based access method is introduced. This access method is a variation of B-trees, but its use results in smaller indices.

## 1. Introduction

An important problem in the development of data management systems has been that of increasing their performance. In this paper we explore the use of low-cost, currently manufactured microprocessors to increase the performance of statistical data management systems.

We define a statistical data management system as one where the use of data is primarily for statistical analysis. In such systems the data are accessed and updated in large quantities. Examples of statistical database applications are sociological and epidemiological studies of data derived from surveys; analyses of economic data for forecasting and modeling; management information systems; and analyses of data resulting from instrumentation of experiments.

Traditional data management research has concentrated on transaction-oriented business systems, such as banking or inventory control systems. [BORA82] contains an introduction to some of the issues in statistical data management research, and points out that business-oriented DBMS do not satisfy the needs of the community of users of data for statistical analysis. [TURN79] describes a data management system for statistical databases.

[1] This work was supported by the Applied Mathematics Sciences Research Program of the Office of Energy Research of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098.

This paper presents a design for a Microprocessor Assist System (MAS), which is a microprocessor-based back-end system that performs a part of the work of a statistical data management system. The MAS is not a database machine because (1) it contains no specialized hardware for data management, only general-purpose microprocessors, and (2) it is not a complete data management system within itself, instead, it is an extension of a front-end resident statistical data management system. However, it is similar to database machines in that it is a system design specialized for data management. [DEWI81] points out that a major problem in the design of database machines is that the cost/performance ratios for the systems are often not considered during the design phase. This paper presents a design, including the cost/performance ratio, and also delineates a general method for determining if the design of a specialized system is practical.

The paper is organized as follows: In Section 2 we propose a general method for the determination of the practicality of a specialized system. Section 3 is a description of the MAS. The MAS is a back-end system that handles functions relating to data access, attribute partitioning and compression for a statistical data management system (SDS). The system design is presented before the functions are described in detail so that the context for discussing the functions' impact on the performance of the SDS can be understood.

In Section 4 both attribute partitioning and compression are shown to be important to a SDS, and to have significant impact on its performance.

Section 5 contains the cost/performance analysis of the a general system compared to the MAS system. It is shown that the MAS has a better cost/performance ratio than a general system.

In Section 6 the use of the MAS in a statistical data management system is discussed. A *category based* access method is introduced. Section 7 is the conclusion.

## 2. Methodology

The following general method to determine the practicality of a specialized system design is proposed:

1) **Determine that the functionality provided by the specialized system is important to the application.**

2) **Show that the cost/performance of the specialized system is better than that of a general system performing the same application.**

3) **Show that the specialized system is easily integrated into the application.**

Step (1) is included because it is important to solve real problems for the application. An example of a solution of a non-problem is associative search for business applications. In analyses in [BEWI81, HAWT81] it is shown that index methods are better suited for retrieving data than associative search methods, if the access is to a single tuple, the media involved is a moving-head disk, and the relation is large. The reason that this is true is that associative search techniques require passing all the data through the read heads; clearly as the amount of data increases the performance advantage of associative search techniques decreases, and eventually index techniques (even with their associated overhead) result in better performance.

Step (2) is necessary because if a design is a *serious* design it is intended to be implemented and used. Designs where the cost/performance is worse than that of general-purpose systems probably will not be built nor used.

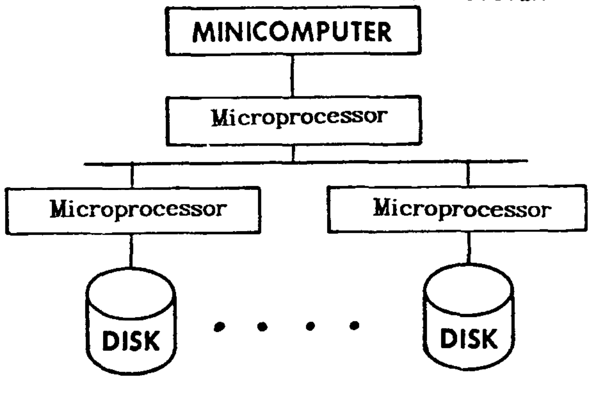
The rationale for Step (3) is to show that the specialized design can be utilized easily in the application. This is a measure of the practicality of the specialized system: that is a natural part of the application as a whole.

In this paper, Section 4, where the importance of attribute partitioning and compression are shown, is included to satisfy step (1). Step (2) is satisfied in Section 5, where the cost/performance of the MAS is shown to be better than that of a general system. Section 6 satisfies step (3) by showing that the MAS can be easily integrated into a statistical data management system.

## 3. MAS Design

The design for the Microprocessor Assist System (MAS) is shown in Figure 1.

FIGURE 1. MICROPROCESSOR ASSIST SYSTEM



The MAS consists of (one or more) trees of microprocessors that are at the bottom level connected one to each disk (the *leaf* microprocessors) and at the highest level (the *root* microprocessor) connected to the front-end computer. A single root microprocessor directs the activities of, and receives data from, its child processors. The tree can be more than two levels, and there may be more than one tree connected to the same front-end. However, for simplicity, in its initial implementation, a two level single tree design is used.

The front-end system is assumed to be a minicomputer with equivalent functionality of a DEC VAX 11/780.

The major functionality of the microprocessors is to implement compression and attribute partitioning techniques for the statistical data management system (SDS) running in the front-end computer. Section 2.3 discusses attribute partitioning and compression; Section 2.5 discusses the part of the MAS in the overall SDS design. This section is concerned with the motivation for the architecture of the MAS.

There are two major differences between this hardware approach and others designed for the management of data: the use of general-purpose microprocessors, and the tree structure of the back-end system.

### 3.1. Use of General Purpose Microprocessors

An example of the type of microprocessor specified in the design is a Single Board Computer currently manufactured by Forward Technology, Inc. This system has a M68000 microprocessor, two 1 Megabyte I/O ports, 256K RAM, and a Multibus[2] interface. Its selling price is $4,000. The processor executes approximately one million instructions per second (1 MIPS).

The leaf microprocessors are in the same position as *data filters* that exist in several database machine designs (e.g. [BANC80, BABB79]), and for the same reason: they operate on data that is read from the disk, pre-processing it before it is sent to the front-end.

Filters are processors that operate at the transfer rate of a disk, on the data as it is transferring from the disk to the front-end system, and perform the operations of restriction, projection, and semi-join. The microprocessors proposed for the MAS are relatively slow (slower than the front-end computer) and do not have the power nor the functionality of general-purpose filters. The microprocessors cannot process data at the speed that it is transferred from the disk because to do so requires at least a 3 MIPS processor to keep up with the transfer rate of the disk [DEWI81]. Instead, they store the data within their memories, and process it more slowly than disk transfer rates.

In the initial implementation, and for the purpose of the analysis in this paper, the MAS *does not* perform general semi-joins, and the only restriction/projections it performs are those immediately related to accessing the data. This is an

[2] Multibus is a registered trademark of INTEL Corporation

example of limited design: we have chosen a few functions for the MAS to do well. If under these conditions it can be shown that the MAS is cost/effective, then expanding its functionality will only make the cost/performance better.

The MAS obtains its performance benefits from the parallel operation of multiple microprocessors. The leaf microprocessors schedule the disk reads, read the required blocks, decompress the data, and send the required data up to the level above; the higher-level microprocessor assembles attributes that are spread across multiple disks (hence multiple microprocessors). Fully assembled tuples are sent from the root microprocessor to the main computer. The fundamental function provided by the MAS is to make compression and partitioned attributes invisible to the front-end system.

Microprocessors are chosen to perform this function rather than attempting to design and implement filters because (1) a microprocessor is sufficient to provide the limited functionality (shown in Section 2.4), and we wish to prove that simply providing the limited functionality will significantly benefit a SDS; and (2) microprocessors of the type described are available to a wide community so that showing their use in a system is of general interest.

## 3.2. Tree-structured Design

The purpose of the MAS is to explore the result of off-loading specific data management functions to a back-end system, and not to explore complex processor interconnection schemes. Therefore, a tree-structured design is chosen because it is architecturally simpler than others used in back-end systems, such as the modified cross-point switch of DIRECT [DEWI79] or the ring structure connection of DBC's post-processing unit [MENO81]. The problem with this tree-structured design is that it does not easily expand to large numbers of microprocessors. Adding multiple leaf processors implies eventually adding more roots (otherwise the root processors would become a bottleneck) therefore forcing the front-end into a complex control operation.

It is not claimed that the MAS is infinitely expandable. Again this is a case of limited design goals: because we target this system for minicomputer applications, the number of disks (hence leaf processors) is small and infinite expansion capability becomes unimportant compared to simplicity of implementation.

## 4. Techniques for Statistical Databases

The purpose of this section is to discuss the issues in step (1) of the design verification methodology: that the functionality provided by the specialized system is important to the application.

In this section two necessary functions of statistical data management systems are discussed. These are compression and attribute partitioning. An estimation is made of the load upon a general-purpose computer for each of the functions.

### 4.1. Compression Techniques

Statistical database management systems must provide compression because often the databases they manage would otherwise be impossible to store on the available disks. [EGGE81, EGGE80] contain discussions of compression schemes applicable to statistical databases. Step (1) is satisfied for compression for SDS by noting that it is widely agreed that the major drawback of compression schemes is that they require CPU time to implement. See [LYNC81] for a discussion of the tradeoffs of compression schemes and a suggestion that microprocessors might be used to alleviate the compression-induced CPU load.

In the remainder of this section we shall develop the number of instructions per page for decompressing a page of data. This number is to be used in the cost/performance section. It is necessarily approximate, because the actual number of instructions per page for decompressing data is dependent on many factors, including the efficiency of the scheme (one hopes that as the data compression rate goes to zero the amount of CPU time to decompress also goes to zero), the particular scheme used, and the type of data. However, to develop an intuition for the cost/performance of a MAS, several assumptions shall be made about the type of compression, the amount of compression achieved, the size and number of attributes per page.

The compression scheme we shall use as an example is run-length encoding. The Socio-Economic, Environmental and Demographic Information System (SEEDIS), the LBL special-purpose system for the manipulation of summary census data, obtains 30% - 60% compression of the incoming data through the use of run-length encoding and removing blanks. In run-length encoding *count*, *value* pairs are stored for values as they are encountered by sequentially accessing the file.

To decompress a *count*, *value* pair, moving the *value* into a decompressed tuple, the system must load the *count* into a register; then each word of the *value* field must be moved to their new locations *count* times. The number of moves is then (number of words in *value*) times (*count*). The minimal number of instructions is then 1 (to load the register) + *count* times words in *value* + 1 instruction per *count* to test that the end of the loop is reached for each compressed value.

For the purposes of this discussion let us assume that a compression rate of 50% is achieved in a relation of 20-byte *values* through the use of run-length encoding. If the *count* fields are two bytes each, the 50% compression implies an average *count* of 2. Assuming that words are 4 bytes each, and that 100 compressed values will fit on a 2K page, then the number of instructions per page is

$$(1 + 2 * 5 + 2) * 100 = 1300 \text{ instructions per page}$$

The instruction-count technique above is the least number of instructions to execute a run-length compression scheme because it does not allow for subroutine calls, error handling, etc. However, it is still significant compared to the processing time required for some statistical queries: in a study of INGRES [HAWT79], a simple statistical query was measured and required about .02 seconds of CPU time per uncompressed page on a 1 MIPS machine (therefore about 20,000 instructions) for the entire processing of the query.

## 4.2. Attribute Partitioning

It is observed in [BORA82, LYNC81, TEIT76] that data for statistical database systems is probably most efficiently stored in a *transposed* file format, although for multi-field accesses such a format exhibits performance degradation.

A transposed file is one where data is stored by field, rather than by record. Thus in a file where the records have several fields each, all the first fields are stored together, then all the second, and so on. Attribute partitioning [HAMM79] is another term for a transposed file scheme within a relational database. As stated in [BORA82], such schemes are useful in statistical database systems because although the relations often contain many attributes, usually only a few are referenced in any one query.

Additionally, attribute partitioning is useful in compression schemes that depend on physical adjacency of identical values [EGGE80, EGGE81, TURN79].

Therefore, we can conclude that attribute partitioning is important to a SDS. The problems inherent in attribute partitioning, and their effect on the performance of the system, are discussed next.

Attributes may be fully partitioned (one attribute per physical file) or grouped (several attributes that are usually accessed together are stored together). In either case performance problems may arise. These are: (1) tuple assembly time: the time to form a tuple from attributes stored in different files; and (2) extra I/O accesses: these occur when a small amount of data is needed, but due to attribute partitioning it is stored on, and must be read from, separate files. These problems are discussed in turn.

First, however, a term must be defined and an example given: a *clustered index* is an index for which the data is physically ordered. This is sometimes called a *primary* index and implies that the relation for which the index exists is physically stored in order according to the clustered index key. If the relation is not partitioned, there can be only one clustered index for it (it can be stored physically only one way) unless the relation is fully replicated.

An example which shall be used in the remainder of this section is **retrieve** *(emp.name, emp.salary, emp.manager)* **where** *emp.dept* = "shoe". (retrieve the names, salaries, and manager's names of all the employees who work in the shoe department). Assume there is a clustered index on *dept*.

### Tuple Assembly

If the attributes of a tuple are not stored together within the same record, some means of associating them and of assembling them must be provided. The two major methods of associating same-tuple attributes are *positional* and *tuple identifiers (tids)*. In the positional method all the attributes for a given relation are stored in the same order. To perform tuple assembly, all attributes with the same logical position are gathered together into the same tuple. In the above example, first the *dept* index would be read to determine the logical positions of the qualifying tuples; then the needed attribute instances would be read. The logical-to-physical translation can be performed where there is no compression by a simple multiplication (length of attribute * position), and where there is compression, by referencing a directory ("attribute *position* begins on page *nn*."), then reading the page sequentially.

A second method of tuple assembly is to assign a unique id to each tuple, then carry the unique id with each attribute. When tuple assembly is required, the attributes with matching tuple ids (tids) are gathered together. This is the method suggested for CASSM [SU75] and DBC [HSIA76] and used extensively in several "inverted file" systems. If the tuple id technique is used, attributes stored on separate files can be oredered differently so that the effect can be almost as good as if there were as many clustered indices as attributes.

In the above example the tids for the qualifying tuples can be read directly from the clustered index on *dept*. Then the associated attributes for which there is an index on tid can be immediately retrieved, and any files for which there is no index on tid must be sequentially read.

Optimal tuple assembly methods for any given system are a function of access patterns and stability of the data: if the data is stable, positional techniques are appropriate; if the data is often accessed according to values of different attributes, tid-based systems may be used even though they are costly in storage space. In any tuple assembly technique the following functions must be performed:

1) Identify attribute instances to be retrieved (via a list of tids, positions, or value ranges).

2) As the attribute value is retrieved from storage, store in a structure such that it can be matched with corresponding attribute values for the same tuple.

3) Pass the assembled tuple to the calling process.

The above three steps require both processing time and space. The space is that needed to store corresponding attributes; the time is required to identify corresponding attributes and to pass the assembled tuple to the calling process.

To determine the effect of tuple assembly on the performance of the system, we need to consider the difference between a system without attribute partitioning and a system with it: the system with it must make at least one data movement and one test instruction per attribute per tuple to be retrieved. Therefore, the minimal effect of attribute assembly is one instruction per word of the attribute, plus one instruction per attribute, for each attribute retrieved. For retrieving 100 tuples with 10 20-byte (5 words) attributes each, the amount of extra time due to tuple assembly is

$$(100 * 10 * 5) + (100 * 10) = 6000 \text{ instructions}$$

6000 instructions, about 3 milliseconds on a 2 MIP machine.

The above analysis results in a minimal cost of attribute assembly because it does not account for space consumption, managing the reading if different data files, etc. Even with the minimal cost associated, however, it is clear that attribute assembly can impact the SDS.

**Block Accesses**

In this section we show that attribute partitioning, although effective for a SDS, degrades badly in some cases. We show that the use of the MAS helps alleviate the degradation.

The fundamental point in achieving performance effectiveness of an attribute partitioning scheme is to determine the access patterns for the users' queries, and to structure the files accordingly. Attributes that are often accessed together are grouped together on the same file to reduce access time. In [HAMM79] heuristics are given for grouping attributes; in [BABA75] a model of system behavior is presented for the case of partitioned files. [CARD75] presents a model for determining both storage required and access times when certain files are inverted.

To show how attribute partitioning effects the performance of a system, we introduce the concept of *efficiency*. The efficiency of an access method is a measure of how much data it caused to be read with respect to the amount of data that was actually needed. In the following section we show that the efficiency of attribute partitioning varies widely according to the structure of the data.

Efficiency can be represented as:

**(EQ 1)**

$$E = \frac{\textit{bytes needed}}{\sum_{i=1}^{\textit{fref}} n_i \bullet \textit{blocking factor}}$$

$E$ = efficiency

*bytes needed* = number of bytes needed to answer query

*fref* = number of files referenced

$n_i$ = number of blocks read from file i

*blocking factor* = number of bytes per block

An efficiency of 1 implies that all the data read was needed. Efficiencies near zero imply highly inefficient access methods: much more data was read than needed.

The *bytes needed* to answer a query is the number of bytes that the SDS requires to process the query and prepare an answer. The number of files referenced, *fref*, is a function of the number of attributes per file and the distribution of the needed attributes across those files, while $n$ is the number of blocks read from each file.

Clearly the efficiency of an access method is dependent on the structures used to organize the storage both in terms of the indices available and the method of partitioning the file. This variation in efficiency is illustrated by the following example.

**Example**

Assume that there are 50 attributes per tuple in a relation, that each attribute is 20 bytes long, that there are 100,000 tuples in the relation, and that data is in 2K blocks. Assume also that a query is run that retrieves five attributes per tuple. The graphs in figures 2 - 5 shows the efficiency of attribute partitioning as the number of attributes per file increases (from 1 to 50), for different numbers of tuples retrieved.

Attribute Partitioning

FIGURE 2

CLUSTERED, OPTIMAL PLACEMENT
RETRIEVE 5 ATTRIBUTES

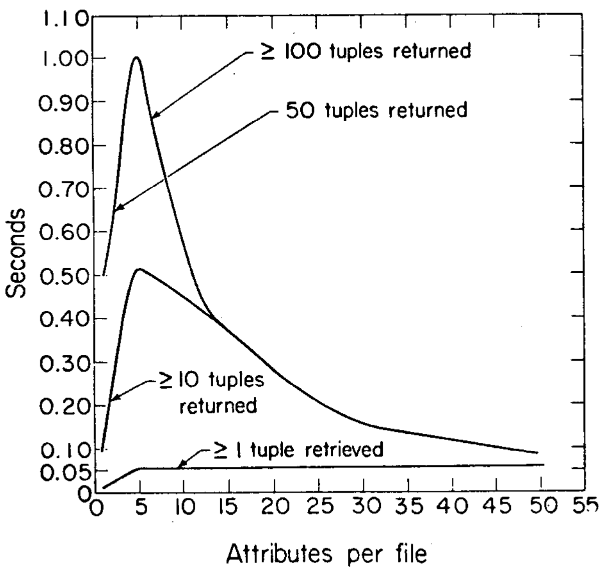


FIGURE 3

CLUSTERED, WORST-CASE PLACEMENT
RETRIEVE 5 ATTRIBUTES

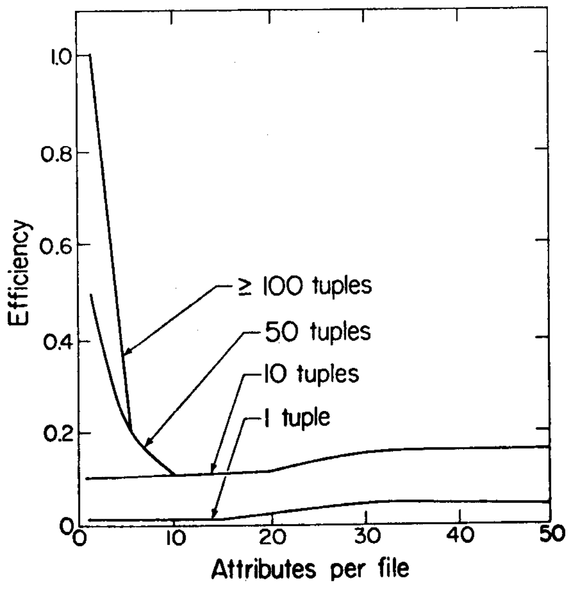


FIGURE 4

CLUSTERED ON ONE ATTRIBUTE ONLY OPTIMAL PLACEMENT, RETRIEVE 5 ATTRIBUTES

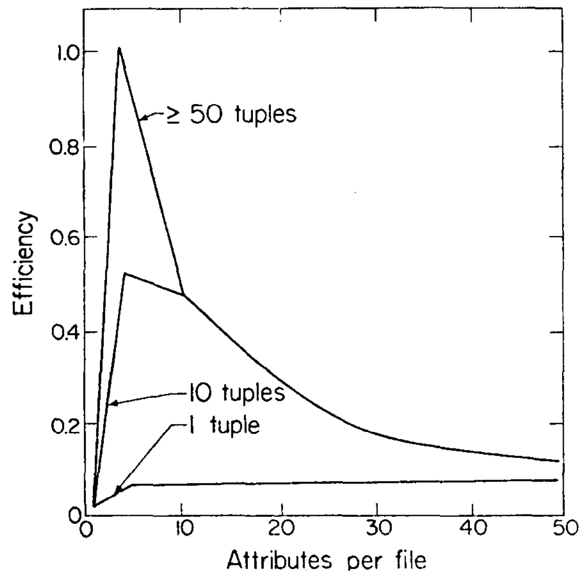
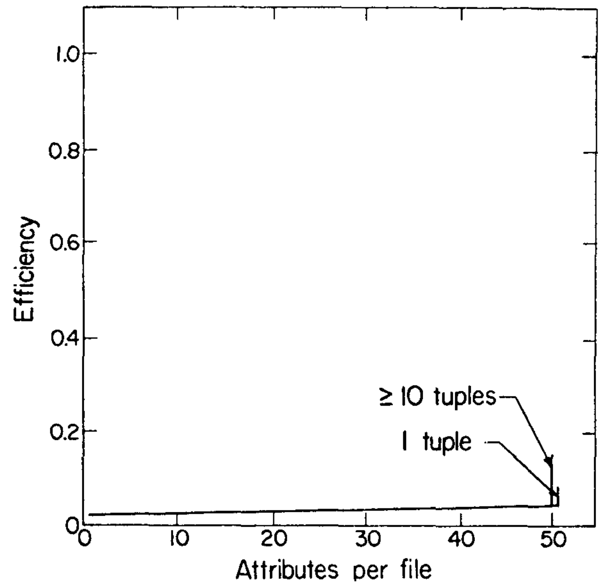


FIGURE 5

CLUSTERED ON ONE ATTRIBUTE ONLY WORST CASE PLACEMENT, RETRIEVE 5 ATTRIBUTES



In each of figures 2 - 5 the horizontal axis is the number of attributes per physical file and the vertical axis is the efficiency (EQ 1). It is assumed that if there are multiple attributes per file they are stored in records (mini-tuples) so that the entire record must be read to obtain any attribute from it.

Figure 2 shows the variation in efficiency in the case where the attributes are optimally placed on the files so that if there are two attributes per file, three files must be accessed to retrieve the needed five; three attributes per file implies that two files must be read; and five attributes per file (and over) implies that only one file must be read. Also in Figure 1 it is assumed that the attributes are clustered such that the SDS can retrieve exactly the blocks that contain the needed attributes by referencing memory-resident indices. In figure 2 we note that retrieving one tuple is never very efficient; the efficiency cannot rise above .05:

*bytes needed* = number of bytes needed to answer query = 5 * 20 = 100

*fref* = number of files referenced = 1

$n_i$ = number of blocks read from file i=1

*blocking factor* = number of bytes per block = 2048

$$E = \frac{100}{2048} = .05$$

The efficiency is a measure of the "wasted data" that the disk must transfer. Therefore an efficiency of 1 is only obtained when all of the data in each block is actually needed by the SDS: this occurs when (1) the attributes are fully partitioned and as many tuples are needed from each block as are stored on the block (in this case, 20 bytes per attribute, 100 attribute values are stored per block), or (2) the data is partitioned such that all the attributes stored on the file are needed by the SDS, and as many tuples are needed from each block as are stored on each block (in this case, for 5 attributes at 20 bytes each, 20 tuples are stored per block, efficiencies near 1 can only be obtained for total number of tuples retrieved near or greater than 20). The efficiencies converge for large values of the number of attributes per file because the ratio of the bytes needed to the total bytes read becomes overwhelmed by the amount of wasted data per block.

Figure 3 shows the situation where the attribute files are clustered, as in figure 2, but the attributes are not optimally placed: five files are always read, until the total number of files is less than five. Again, the efficiency for the single tuple case is always low. The only efficient storage structure is the fully partitioned case; all others must pass too much wasted data.

In figure 4 it is assumed that the attributes are optimally placed, but that only one has a clustering index. In order to assemble tuples, the other four attributes must be sequentially read if they are on other files. Clearly in this case the only possible efficient structure is where all five attributes needed are on the same file, and where the total number of tuples retrieved is greater than or close to the number stored on per block.

Figure 5 shows the worst case: the attributes are not optimally placed, and only one has a clustering index. In this case, if there is more than one attribute file each of the remaining files must be read entirely to assemble the tuples. When there is only one file, the amount of wasted space due to storing all 50 attributes on the same file causes the efficiency to remain very low.

**End Example**

The above example shows that partitioned attributes, while not particularly effective for single-tuple accesses, is a well suited technique for applications where the data is accessed in large quantities, as in statistical data management systems.

The efficiency of the partitioning scheme is related to the use of the data; it may be assumed that the SDS does not optimally partition the data for all cases. Therefore in some instances less efficient methods may have to be used. It is clear that attribute partitioning, while optimal for a SDS, can result in a heavy load upon the resources of the system when less efficient accesses must be used.

A primary purpose of the MAS is to make attribute partitioning invisible to the front-end system: the front-end simply requests the data it needs, and the MAS provides it in decompressed, assembled tuples. The effect of less-efficient accesses is that the total number of pages to be read is greater; in that case the MAS, by off-loading the data access activities and relegating them to multiple processors, is able to soften the effect of inefficiencies in the access methods.

## 5. Cost/Performance

In order to justify the use of a back-end system to increase the performance of a SDS, a model of the system must be developed with and without the back-end, and the performances and costs compared.

The response time of a SDS without the MAS can be represented as:

**(EQ 2)**

$$R = P + a + m \bullet A + \sum_{i=1}^{i=fref} n_i \bullet (c+t)$$

$R$ = response time
$P$ = CPU time to process tuples
$a$ = CPU time to decompress and assemble tuples
$m$ = number of unoverlapped seeks
$A$ = access time for the disk
$fref$ = number of files referenced
$n_i$ = number of blocks read from attribute file i
$c$ = CPU time to schedule and read a block of data
$t$ = transfer time for a block of data

Response time is a function of the number of unoverlapped seeks, the number of files referenced and number of blocks read from each file, the CPU and transfer times for each block and the CPU time to process the assembled tuples.

In the case of a system that includes a MAS, the response time is:

**(EQ 3)** $$R = P + \frac{a'}{k} + m \bullet A + \frac{\sum_{i=1}^{i=fref} n_i \bullet (c'+t')}{k}$$

$$+ nans \bullet (c'+t') + 2 \bullet cp \bullet k + C \bullet nans + C$$

$R$ = response time
$P$ = CPU time to process tuples, in front-end
$a'$ = micro CPU time to decompress and assemble tuples
$k$ = number of leaf microprocessors
$m$ = number of unoverlapped seeks
$A$ = access time for the disk
$fref$ = number of files referenced
$n_i$ = number of blocks read from attribute file i
$c'$ = micro CPU time to schedule and read a block of data
$t'$ = micro transfer time for a block of data
$nans$ = number of blocks of data in answer
$cp$ = micro-to-micro communication time
$C$ = front-end-to-micro communication time

The response time has three major components: front-end processing time, I/O time, and MAS processing time. The front-end processing time is sum of the time to communicate from the front-end to the root micro ($C$), the time to receive the required data from the root micro ($C$ times the number of blocks on the answer, $nans$) and the processing time in the front-end, P. The I/O time is the same as in the standard system, $m \bullet A$.

The processing time in the MAS is composed of the following: One of the micros in the MAS must serially receive data from the leaf micros and send it to the front-end: the time to perform that function is the time it takes to receive a data block ($c' + t'$) times the number of blocks in the answer, $nans$. The root micro must send commands to the leaf micros: assuming one command per query, the response time addition due to the root-leaf command is $cp$ times the number of micros ($k$) times 2 (one send, one acknowledge). The MAS leaf processing time is the sum of the time to assemble and decompress the tuples, $a'$ and the time to process the pages as they are read,

$$\sum_{i=1}^{i=fref} n_i \bullet (c'+t')$$

divided by the number of leaf processors executing the query. If we make a convenient assumption, that all the leaf processors are busy executing the command, then the assembly, decompression, and block processing times can be divided by the number of leaf processors, $k$.

In comparing (EQ 2) and (EQ 3) it is apparent that the time to process the assembled, decompressed tuples in the front-end ($P$) and the I/O time ($m \bullet A$) are unchanged by the addition of the MAS. They will therefore be dropped from comparisons of the two systems.

To compare the two systems we will consider two real computer systems available: for the front-end, a VAX 11/780 (instruction speed about 2 MIPS) and for the microprocessors, Single Board Computers, described in Section 3.

To compare the response times of the systems we will fix a number of values for the above variables:

$a$ = CPU time to decompress and assemble tuples = .004 sec / blk
$k$ = number of leaf microprocessors = 4
$fref$ = number of files referenced = 4
$c$ = CPU time to schedule and read a block of data = .001 sec
$t$ = front-end transfer time for a block of data = .00005 sec

$t'$ = micro transfer time for a block of data = .0005 sec
$cp$ = micro-to-micro communication time = .002 sec
$C$ = front-end-to-micro communication time = .001 sec

The front-end CPU time, $\alpha$, is the time to decompress and assemble one 2K byte data page given that the processor is a 2 MIP processor is the time calculated in the sections concerning decompression and assembly. The CPU time to schedule and read a block of data is the measured time for a VAX 11/780 minicomputer running UNIX[3]. The transfer time for a block on both the front-end and microprocessor systems is the time that the system is completely unavailable to do anything else while receiving the data. This time is a function of the I/O bandwidth, the memory cycle times, and the bus architecture. For a VAX 11/780, that time is .00005 sec per 2K block; for a Single Board Computer, that time is .0005 sec / 2K block.

The time to communicate from the front-end to the root micro is taken to be .001 sec for the first analysis; this is the same communication time as is necessary to communicate with a disk. In the next section that time is varied to determine the effect of higher communication times. The time to communicate from one micro to another is fixed at .002 sec, twice the time for the front-end, because the micros have half the processor speed of the front-end.

Figure 6 shows response time, in seconds, as a function of the number of tuples retrieved. The three lines are (lower) the MAS-VAX system described above (1 MIPS micros, 2 MIPS front-end); (middle) a standard VAX system; and (upper) a 4-leaf MAS-VAX system where the micros operate at 1/5 the processing speed of the VAX (with appropriate changes to the communication times as well as micro processing and transfer times).

Figure 6 shows that the multiprocessing capability of the microprocessors allow them to out-perform the general purpose system as long as the aggregate processing speed of the microprocessors is larger than that of the general system. The MAS has a response time less than that of the general system because each disk is transferring data into its own microprocessor, which acts independently of the front-end system, sending only the required data up to the root micro, where only assembled tuples are sent to the front-end. The MAS therefore relieves the general system of all functions related to tuple assembly and compression, resulting in a performance benefit as long as the MAS can perform those functions faster than the general system.

[3] UNIX is a trademark of Bell Laboratories

FIGURE 6

4-ATTRIBUTE RETRIEVE, CLUSTERED

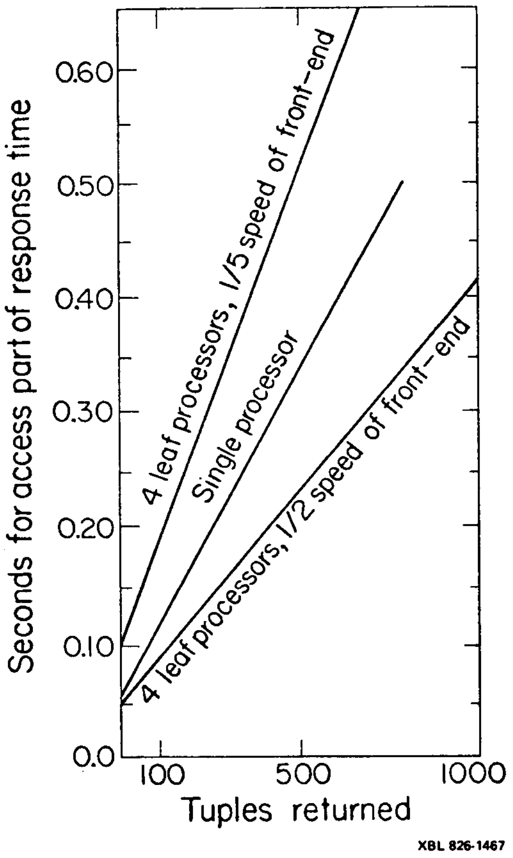


FIGURE 7

4-ATTRIBUTE RETRIEVE, CLUSTERED

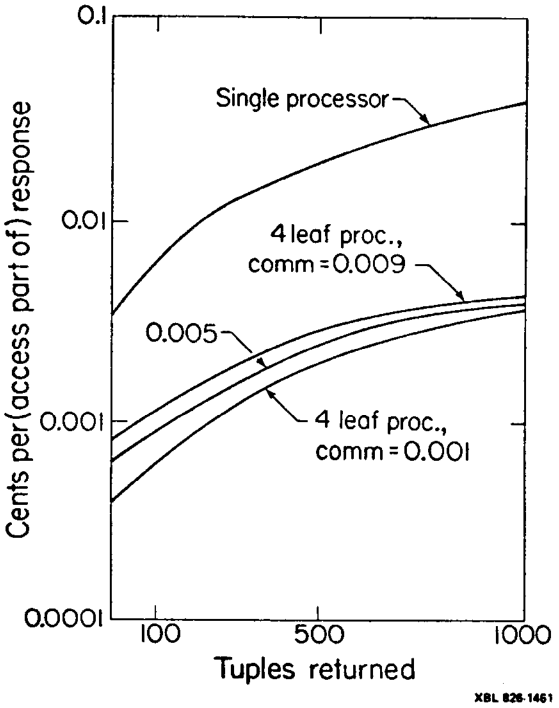


Figure 7 shows a comparison of the two systems from another viewpoint: the cost/performance. If we assume that the VAX cost $200,000, to be amortized over 2 years, then a second of processing time on the VAX costs .06 cents. The micros to be used for the MAS (the single-board computers described above) cost $4,000 each. Assuming another $2,000 each for communication equipment, the 4-leaf, 1-root MAS described above costs $30,000. Amortizing that over 2 years gives a cost per second on the MAS of .008 cents. Figure 7 shows the cents for the access-method part of the response time of a four-attribute query, where the number of tuples retrieved is varied from 1 to 1000. The lower curve is plotted with the assumption that the communication time from the front-end to the micro is .001 sec., the same as it is between the front-end and a disk. The upper curve is the cost per response of the VAX alone.

It is not clear that communicating with the MAS is the same as communicating with a disk, so the middle curves are plotted with the communication time as .005 sec per block (bottom) and .009 sec. (top). In all cases the MAS was much more cost-effective than the VAX alone, even when the front-end to back-end communication time was increased.

The cost/performance graph, Figure 7, was plotted with the assumption that the data is optimally placed and clustered. If the data is not clustered, then the amount of processing time to read the data goes up, and the cost for the general system increases much higher than the same cost using the MAS. If the data is not optimally placed, the response time for the MAS will be slower for that particular query than if the data were optimally placed (and all processors working on the same query). However, we can assume that the MAS can be at the same time executing requests for data from other queries, so that the number of leaf micros in use is nearly always the number on the system.

We therefore conclude that the MAS is a cost-effective system that will improve the performance of a SDS.

## 6. Using the MAS in a System

The MAS is used as follows: the SDS receives a query from a user and determines which (relations, attributes) are needed. It passes to the MAS a data request: "get *attribute-list*, *relation* for *attribute1 between xx and yy*, *attribute2 between zz and qq*, ...". The root micro receives the request and sends the appropriate requests to the child micros; as they send the data back, the root micro assembles the tuples and sends them to the SDS. To process updates, the front-end sends the MAS the assembled tuples, with the relations and attributes identified; the MAS performs the partitioning and compression. The MAS simply requires a high-level interface within the SDS to be integrated into the system, which it is clear can be accomplished in a straight-forward manner. There is a problem, however: the MAS micros have a limited amount of memory in which to store indices. If normal indexing methods are used, the MAS may be very inefficient in accessing data. In the remainder of this section we discuss an indexing method that will result in smaller indices.

### Category Access Methods

[BORA82, TEIT76] observed that the major method of accessing data in statistical databases is by *category* rather than by the individual record keys often used in business data management systems. A category is a major horizontal partition of the data, usually expressible by simple boolean functions (e.g., *age<20*, *20<=age<=40*, *age>40* ). Data is separated into major categories in order to provide meaningful analyses.

We assume that any data management system designed for the statistical analysis of data will contain methods for the user to specify category attribute values. The following is a suggested physical implementation of category-based access methods.

Assume that the attributes are fully partitioned so that each category attribute can be sorted separately. Then to build the category index there are two steps:

(1) sort the attribute-based partition, compressing if possible
(2) build a B-Tree like index which consists of pointers (beginning and end) to the user-specified category boundaries for the attribute.

The following is an example of this method:

**Example**

Assume the attribute *time* is declared to be a category attribute, the boundaries specified by *time <= 00999*, *01000 <= time <= 30000*, *time >= 30001*. and assume that there are one million tuples in the underlying relation. The *time*, *tid* pairs are sent (in tuple id (tid) order) to the index creation process, which sorts the attribute by time, and as each boundary is reached enters the page number in the index. Then example data pages are:

**Figure 8. Data Page**

| left | right | up |
|---|---|---|
| 04567 | 01395 | 012987 |

| time | tid |
|---|---|
| 01295 | 013456 |
| 01295 | 015689 |
| 01296 | 016790 |

The entries *left*, *right* are pointers to the left and right sibling data pages, *up* points to the parent index page, *time* and *tid* are in binary representation.

The index page contains the usual B-Tree pointers and the following:

**Figure 9. Index Page Data Pointers**

| value1 | value2 | start | end | number |
|---|---|---|---|---|
| -1 | 999 | 2456 | 2678 | 500000 |
| 10000 | 30000 | 2679 | 5926 | 100000 |
| 30001 | 9999999 | 6891 | 7345 | 400000 |

*value1*, *value2* represent the beginning and ending values of the attribute; ***start***, ***end*** are the page numbers for the beginning and ending pages. It is important that the ending page be kept track of so additions can be quickly made. ***number*** denotes the number of tuples within the partition; this information is useful for query optimization schemes.

Note that since *time* can be expected to have many unique values over the one million tuples in the relation, any ordinary record-oriented index structure must be very large. The effect of the category-based access method is to reduce the size of the index because pointers are only retained for category boundaries.

**End Example**

In step (2) above, the "create index" process marked the category boundaries on the index page and was done. It can be expected, however, that different users will want to see different category boundaries. These can be implemented by the SDS forming "intermediate" categories whose unions form the categories required for each user. For example, suppose in the above example a second user defines the *time* boundaries to be: *time* <= *500*, *501* <= *time* <= *20000*, *time* >= *20001*. Then an implementation of the category index is:

**Figure 10. Index Page Data Pointers**

| value1 | value2 | start | end | number |
|---|---|---|---|---|
| -1 | 500 | 2456 | 2137 | 200000 |
| 501 | 999 | 2142 | 2697 | 200000 |
| 10000 | 20000 | 2679 | 5926 | 100000 |
| 20001 | 30000 | 6131 | 6233 | 100000 |
| 30001 | 9999999 | 6891 | 7345 | 400000 |

If both boundaries are presented to the SDS at the same time, the "create index" command has only the task of splitting up the index into component parts such that both boundary specifications are satisfied. A more difficult problem occurs when a new set of boundaries is specified for an existing category attribute. There are two cases: where the data is sorted within the categories, and where it is not.

### 6.1. Un-sorted Data

In the above implementation it is assumed that the data within categories does not need to remain sorted: that new items can be added at the end of the category, deleted items marked, and occasionally (in the middle of the night) "clean-up" algorithms run to re-form the attribute files that have undergone extensive updates. Rewriting extensively updated files is necessary because physical sequentiality of logically sequential data must be maintained to minimize disk head movement [HAWT79].

This approach works well for statistical databases where the data within the attribute files does not have to remain sorted because the analysts are only concerned with major partitions of the data, not with the specific ordering within the partitions. Then, if new categories are created, the "create index" process must decide on the new partitions, sort the data within the partitions to be changed, and re-form the index page.

### 6.2. Sorted Data

If the application is such that the data must remain sorted, then either a binary insertion technique (for one-at-a-time updates) or a sort of incoming data, then merge with the categories (for multiple additions at once) must be performed to keep the data in the partitions sorted. Then new categories can be added by simply implementing several sets of category indices (e.g., one per user), or by re-forming the old one.

### 6.3. Benefits

The benefit of the MAS is to off-load a portion of the work of the SDS. The category-based access method is of use to both a general-purpose and a MAS-type system; its advantage is that the index is small, representing only the necessary data.

## 7. Conclusion

The general method proposed to determine if a system design is practical is:

1) **Determine that the functionality provided by the specialized system is important to the application.**
2) **Show the cost/performance of the specialized system is better than that of a general system performing the same application.**
3) **show that the specialized system is easily integrated into the application.**

In Section 4 both attribute partitioning and compression are shown to be important to a SDS, and have significant impact on its performance. Thus, Step (1) is satisfied in Section 4.

In Section 5 it is shown that the MAS has a much better cost/performance ratio than a general system; therefore Step (2) is satisfied.

In Section 6 the use of the MAS in a statistical data management system is discussed. A category-based access method is introduced. This is a variation of B-trees. In B-trees all the unique values of the records to be indexed are stored in the leaves of the B-tree. It is observed that in statistical applications users are interested in broad categories, not unique records, so that the category-based indices are smaller, and therefore more easily stored in the small memories of the microprocessors of the MAS. Since the interface to the MAS is a simple, high-leveled one, it is argued that the MAS is easily integrated into the SDS, thus satisfying Step (3).

Therefore, we have presented a design for a microprocessor system that is a practical solution to some of the problems in statistical data management.

*Acknowledgements:*

The author would like to gratefully acknowledge the help of the following: for their support and careful reading of this paper: Arie Shoshani and John L. McCarthy of LBL and Michael Ubell of Britton-Lee.; the suggestion for the need for a currently-implementable system: Professor Lawrence Rowe, of UC Berkeley; and Michael Tiberio of Lincoln Laboratories for pointing out the problem of updating record-oriented indices.

## BIBLIOGRAPHY

[BABA77] Babad, J.M., "A Record and File Partitioning Model", Commun. of the ACM. Jan., 1977

[BABB79] Babb, E., "Implementing a Relational database by Means of Specialized Hardware," ACM Transactions of Database Systems, Vol. 4, No. 1, March 1979, pp. 1-29.

[BANC80] Bancilhon, F., and M. Scholl, "Design of a Backend Processor for a Data Base Machine", Proceedings, ACM Sigmod International Conference on the Management of Data, Santa Monica, California, May 1980.

[BORA82] Baoral, Haran, David J. DeWitt and Doug Bates, "A Framework for Research in Database Management for Statistical Analysis", Computer Sciences Tech. Report #465, University of Wisconsin-Madison, Feb. 1982, to appear in Sigmod, 1982.

[CARD75] Cardenas, A.F., Analysis and Performance of Inverted Data Base Structures:, Commun. ACM., May, 1975 (with a correction in Yao, S.B., "Approximating Block Accesses in Database Organizations", Commun. ACM., April 1977.)

[DEWI78] DeWitt, David, "DIRECT - A Multiprocessor Organization for Supporting Relational Data Base Management Systems, "Proc. Fifth Annual Symposium on Computer Architecture, 1978.

[DEWI81] DeWitt, David and Paula Hawthorn, "A Performance Evaluation of Database Machine Architectures", Proceedings, IEEE and ACM 7th conf on Very Large Databases, Cannes, France, 1981.

[EGGE80] Eggers, Susan and Arie Shoshani, "Efficient Access of Compressed Data", Proceedings, IEEE and ACM 6th conf on Very Large Databases, Montreal, Canada, 1980.

[EGGE81] Eggers, Susan, Frank Olken and Arie Shoshani, "A Compression Technique for Large Statistical Databases", Proceedings, IEEE and ACM 7th conf on Very Large Databases, Cannes, France, 1981.

[HAMM79] Hammer, M., and B. Niamir, "A Heuristic Approach to Attribute Partitioning", Proceedings, ACM Sigmod International Conference on the Management of Data, Boston, 1979.

[HAWT79] Hawthorn, Paula, "Evaluation and Enhancement of the Performance of Relational Database Management Systems", ERL Memo No. M79/70, Electronics Research Laboratory, UC Berkeley, November, 1979.

[HAWT81] Hawthorn, Paula "The Effect of Target Applications on Database Machines", Proceedings, ACM Sigmod International Conference on the Management of Data, Ann Arbor, Michigan, 1981.

[HSIA76] Hsiao, D.K., and K. Kannan, "The Architecture of a Database Computer - Part II: The Design of Structure Memory and its Related Processors," National Technical Information Service Number AD/A035 178

[LYNC81] Lynch, Clifford A. and Edwin Brownrigg, "Application of Data Compression Techniques to a Large Bibliographic Database", Proceedings, IEEE and ACM 7th conf on Very Large Databases, Cannes, France, 1981.

[MENO81] Menon, M. J. and David K. Hsiao, "Design and Analysis of a Relational Join Operation for VLSI", Proceedings, IEEE and ACM 7th conf on Very Large Databases, Cannes, France, 1981.

[SU75] Su, S.Y.W. and G.J. Lipovski, "CASSM: A Cellular System for Very Large Data Bases", Proceedings of the VLDB, 1975, pp. 456-472.

[TEIT76] Teitel, Robert F., "Data Base Concepts for Social Science Computing", Proc., Computer Science and Statistics 9th Annual Conference on the Interface, Harvard, 1976; Proceedings available from: Prindle, Weber and Schmidt, Inc., 20 Newbury St., Boston, MA 02116.

[TURN79] Turner, M.J., R. Hammond and P. Cotton, "A DBMS for Large Statistical Databases", Proceedings, IEEE and ACM 5th conf on Very Large Databases, Rio de Janeiro, Brazil, 1979.