Algorithms for Loading Parallel Grid Files *

Jianzhong Li[†] Doron Rotem[‡] Jaideep Srivastava[§]
Information and Computing Science Division
Lawrence Berkeley Laboratory, Berkeley, CA 94720

Abstract

The paper describes three fast loading algorithms for grid files on a parallel shared nothing architecture. The algorithms use dynamic programming and sampling to effectively partition the data file among the processors to achieve maximum parallelism in answering range queries. Each processor then constructs in parallel its own portion of the grid file. Analytical results and simulations are given for the three algorithms.

1 Introduction

In this paper we study efficient multiprocessor algorithms for initial loading of large existing data files into a spatial data structure. This work is motivated by scientific and GIS applications such as climate modeling, seismic studies and physics experiments, where large data files are generated by simulation programs or by collecting recorded measurements of a large number of sensors. Various subsets of this large volume of multidimensional data, which often resides on robotic tape libraries, must be loaded into an appropriate disk based spatial structure for the purpose of analysis, querying and visualization. In such applications, loading of a new subset is performed frequently as users shift their focus of attention to different regions of interest within the original file. Additional cases where fast loading is important include:

- Recovery - Reconstruction of a spatial structure after recovery from a disk failure may involve loading of large files from backup tapes or disks.
- Reorganization - Dynamic spatial data structures may suffer from deterioration of storage utilization over time due to large directory and sparsely filled data pages. The algorithms described here can be used to restore the storage utilization to an acceptable level while reloading the file.

There are quite a few papers in the literature dealing with efficient retrieval of spatial data structures on parallel architectures [LSR92,GAK90,FKK91]. The objective of these research works is to find methods for distributing the data among processors in order to balance the work involved in computing the answer to a query. However, to the best of our knowledge, very little research work has been done on the initial loading of the spatial data structure which may become a serious bottleneck in the above mentioned applications. In the case of one dimensional data, there has been some work on fast loading of B-trees [ROS81] and many commercial database systems have bulk loading utilities which allow fast construction of index structures while loading the database.

Many dynamic spatial data structures are based on some partitioning of the space into regions with the property that the data that belongs to one region fits on a single disk page. The partitioning points along each dimension and the assignment of disk pages to regions are maintained in some type of directory consisting of one or more levels.

Growing a spatial data structure dynamically by inserting individual records one at a time may be a relatively slow process as each disk page overflow may require updating the directory, as well as mov-

*Supported by the U.S. Department of Energy under Contract DE-AC03-76SF00098.

[†]On leave from Heilongjiang University, P. R. China.

[‡]also with Department of MIS San Jose State university

[§]Is with the Computer Science Department, University of Minnesota at Minneapolis.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0347...$1.50

ing some data from the overflowed page into a newly created one. Furthermore, in some data structures, a single insertion may cause several disk I/O's due to a chain of page splits. In this paper we show that in case the data already exists in some format, a much faster bulk loading process is possible.

The idea behind our algorithms is to perform some precomputation on the input file and use the results to determine optimal or close to optimal partitioning points along each dimension such that the file to be loaded can then be partitioned among the processors. This phase is called logical partitioning and its main goal is to achieve a balanced work load among the processors for computation of answers to range and partial match queries. The second phase, called physical partitioning, is performed in parallel by all processors. Each of the processors performs some precomputation on the subfile assigned to it in the previous phase. The results of this precomputation are then used in constructing their subdirectories and loading their portion of the data.

In this paper we are concerned mainly with efficient parallel loading of grid files. However, similar techniques can be used to parallelize the loading of other directory based spatial data structures.

The paper is organized as follows. In Section 2 some background and terminology are introduced. In Section 3 we describe the parallel grid file structure. In Section 4 three parallel loading algorithms are described. In Section 5 we analyze the overflow generated by the sampling procedure. In Section 6, we present simulation results and comparisons between the algorithms. Finally, in Section 7 some conclusions and directions for future research are given. Proofs for the lemmas and theorems are provided in the Appendix.

2 Terminology and Background

2.1 Parallel Computing Architecture

For ease of presentation in this paper, we assume a shared nothing multiprocessor architecture, where each processor has one dedicated disk [STO86]. The processor interconnection is a hypercube which is one of the most versatile and efficient networks yet discovered for parallel computation. It is well suited for both special-purpose and general-purpose tasks, and it can efficiently simulate any other network of the same size. It can solve arbitrary message-routing problems in $O(log_2 N)$ steps where N is the number of processors. A processor with its associated private memory and dedicated disk is called a *processing node*. We assume that the file to be loaded is initially stored at a processing node, called the *coordinator*.

2.2 Multidimensional Database Files

Let D_i ($1 \leq i \leq d$) be an ordered set. *A d-dimensional file scheme on D_1, ..., D_d* consists of d attributes A_1, A_2, ..., A_d where D_i is defined to be the domain of attribute A_i. *An instance of a d-dimensional file scheme* is a non-empty set of records, where a *record* is an ordered d-tuple $(r_1, r_2, ..., r_d) \subseteq D_1 \times D_2 \times ... \times D_d$ and r_i is the value of the record on attribute A_i. In the rest of the paper we simply call both a d-dimensional file scheme as well as its instance a *d-dimensional file* or *file*. For simplicity, we assume that all files are subsets of the unit space $S = [0,\ 1)^d$. However, one should note that the algorithms in this paper are not limited to the space S.

2.3 Grid File Structure

A grid file is a multi-dimensional file structure [NIE84] in which records of a d-dimensional file are represented as points in a d-dimensional space. The space is divided into a set of hyper-rectangles called grid blocks, each of which is assigned to a disk page. Several grid blocks forming a hyperrectangle may be assigned to the same disk page. A grid file consists of a *grid directory* and a space partitioned by a *grid partition scheme*.

A *grid partition scheme* can be obtained by imposing intervals on each axis and partitioning the space into blocks. The grid partitioning scheme is composed of d one-dimensional arrays called *linear scales*, each of which defines a division of a dimension of the space. During the operation of a grid file system the underlying partition of the space may need to be modified in response to splitting an interval or merging of two adjacent intervals.

The task of the *grid directory* is to assign grid blocks to disk pages. A grid directory consists of a dynamic d-dimensional array whose elements (pointers to data pages) are in one to one correspondence with the grid blocks of the partition. When the assignment of grid blocks to pages changes, an update of the directory is needed.

In the grid file structure, an exact-match query needs only two page accesses: the first access to the directory and the second to the correct data page. The grid file can process range queries with respect to all attributes efficiently because it can preserve the order defined on each attribute domain so that records being near in the domain of any attribute are likely to be on the same page. Partial match queries, which are a special case of range queries, are efficiently supported as well.

2.4 Multidimensional Data Declustering for Grid Files

Multidimensional declustering is a partitioning scheme which determines which part of the data will reside on each processing node. As the size of an answer to a range query may be very large, the objective of multidimensional declustering is to achieve balance among the processors in computing the results of such queries.

3 Parallel Grid File Structure

A parallel grid file system is a two level structure. One is a *logical parallel grid file structure* and the other is a *physical parallel grid file structure*. Most of the concepts in this section were first introduced and proved in [LSR92], in order to make this paper self contained we discuss here briefly the main results of that work.

3.1 Logical Parallel Grid File Structure

Given a d-dimensional file F, this structure describes the *logical grid partition* of F and the assignment of the *logical grid blocks* resulting from the partition to processing nodes. The capacity of a logical grid block is not necessarily equal to the size of a disk page. It is determined by the expected "area" of a range query as explained in [LSR92]. In general, if range queries typically cover a small area compared to the space covered by the grid file, smaller logical blocks are needed to guarantee maximum parallelism.

3.1.1 Logical Grid Partitioning

Given a d-dimensional file F and capacity C of a logical grid block, the primary objective of the logical grid partitioning is to partition F into grid blocks by dividing all D_i's into intervals such that the data skew among blocks is minimized. In this paper, the metric used to measure the logical data skew of a partitioning scheme is simply the total absolute deviation of all partition blocks from C, the logical capacity of a grid block. This measure assigns an equal penalty for overflow and underflow. As we typically do not allow overflow in grid files, we use total overflow as the metric for measuring physical data skew. (See next section for a formal definition). The algorithms can be adapted to minimize objective functions based on other metrics such as worst case deviation.

A simple method of declustering F is to divide each dimension into equally spaced intervals. However, if F is not uniformly distributed, this may lead to a large imbalance between the number of points in each block. Our strategy is to partition S into $\prod_{i=1}^{d} n_i$ logical grid blocks where n_i's are determined by partitioning algorithms which are sensitive to the distribution of F. These partitioning algorithms will be discussed in detail later. Here we only provide an overview of the logical grid partitioning.

The partitioning of F is represented by d vectors, called *partition vectors*. The ith partition vector describes the partition of the ith domain D_i, which is defined as $V_i = (p_1, p_2, \ldots, p_{n_i-1})$, where $0 < p_1 < p_2 < \ldots < p_{n_i-1} < 1$ are the points that partition D_i into n_i intervals

$$[0, p_1), [p_1, p_2), \ldots, [p_{n_i-1}, 1),$$

for $1 \leq i \leq d$.

The intervals on each dimension are numbered from 0 to $n_i - 1$. The kth interval of D_i is denoted by I_{ik}, for $0 \leq k \leq n_i - 1$. We define the *coordinate of the interval* I_{ik} to be k. Hereafter, we use $[l_{ki}, h_{ki})$ to represent interval I_{ki}. Consider two intervals I_{ik_1} and I_{ik_2}. If $k_1 < k_2$, then every point in interval I_{ik_1} is less than every point in interval I_{ik_2}. After logical grid partitioning, F is divided into $\prod_{i=1}^{d} n_i$ *logical grid blocks*, where a logical grid block is the cross product of d intervals, i.e.,

$$[l_{1k_1}, h_{1k_1}) \times [l_{2k_2}, h_{2k_2}) \times \ldots \times [l_{dk_d}, h_{dk_d}),$$

where, $0 \leq k_j \leq n_j - 1$, $1 \leq j \leq d$. We define the *coordinate of the logical grid block* above to be $(k_1, k_2, \ldots, k_d)$. The logical grid block with coordinate $(k_1, k_2, \ldots, k_d)$ is represented by $R_{k_1, k_2, \ldots, k_d}$. In the rest of the paper, we use capital letters X, Y, ... to represent the coordinates of regions in S and

x, y, ... to represent the coordinates of the records in F, i.e. points in space S.

3.2 Assignment of Logical Grid Blocks

First, we define an allocation function, called *Coordinate Modulo Declustering* (CMD), which determines the processing node to which a logical block is assigned. It is defined as:

$$CMD(X_1,\ X_2,\ ...,\ X_d) = (X_1 + X_2 + ... + X_d) \ mod\ N.$$

The region $(X_1,\ X_2,\ ...,\ X_d)$ is assigned to processing node $CMD(X_1,\ X_2,\ ...,\ X_d)$, where the N processing nodes are numbered 0, 1, ..., $N-1$.

After the logical grid partitioning and the allocation of the logical grid blocks to processing nodes, we have a ***logical parallel grid file structure*** for the file F, denoted by $(F,\ (V_1, ..., V_d),\ CMD)$ where V_i is a $(n_i - 1)$-dimensional partition vector, for $1 \leq i \leq d$.

3.3 Physical Parallel Grid File Structure

Given a d-dimensional file F and its logical parallel grid file structure, the physical parallel grid file structure describes the organization of the subfiles of F on processing nodes. In the following, let F_i and S_i represent the subfile of F and the subset of S allocated to processing node i for $0 \leq i \leq N-1$.

In general, any uniprocessor spatial data structure can be used in conjunction with the logical parallel grid file to implement a complete parallel file structure. In this paper, we use the uniprocessor grid file structure to organize F_i on processing node i. The regions in S which are assigned to F_i lie along parallel diagonals in the logical parallel file structure, and form a highly non-uniform subspace. It is therefore not efficient to organize F_i by directly using a grid file structure.

To solve the non-uniformity problem, we define a coordinate transformation function on S_i to map S_i into a d-dimensional rectangle and map F_i into a uniformly distributed file in this rectangle. In the 2-dimensional case, the transformation we define amounts to collecting from each column only those regions which are assigned to processing node i collapsing all the other regions.

4 Parallel Loading Algorithms

In this section, we develop three parallel algorithms for loading parallel grid files. The algorithms differ by the cost of precomputation they perform on the initial file. Given a d-dimensional file F, the following three tasks need to be performed by a loading algorithm:

1. Create the logical parallel grid file structure for F;

2. Create the physical parallel grid file structure for each F;

3. Load the data of F into the parallel grid file resulting from steps 1 and 2.

In order to guarantee load balancing on processing nodes and high parallelism and low I/O cost for query processing, the objective of the algorithms is to minimize the ***total data overflow of the physical grid blocks*** and the ***total data skew of the logical grid blocks***. The ***total data overflow of the physical grid blocks*** is defined as $\sum_{i=1}^{P} MAX(0, C_i - c)$, where P is the number of disk pages, C_i is the size in bytes of the ith physical grid block and c is the capacity of disk page. The ***total data skew of the logical grid blocks*** is defined as $\sum_{i=1}^{LP} \mid LC_i - C \mid$, where LP is the number of logical grid blocks, LC_i is the size in bytes of the ith logical grid block, and C is the capacity of a logical grid block.

4.1 Two Phase Optimal Algorithm

In this section, we present a two phase optimal algorithm, called TOPLOADING. The algorithm performs logical and physical partitioning of file F in two phases. In the logical partitioning phase, it partitions F into $h \times v$ logical grid blocks, where h and v are fixed and depend on the properties of the queries on F. Based on a theorem proven in [LSR92], it is beneficial to have at least one of them be of the form kN for some integer k so that the data distribution among processing nodes will be balanced. After the logical partitioning phase, the logical grid blocks are allocated to processing nodes according to the CMD function, and the partition F_i of F is sent to processing node i, for $0 \leq i \leq N-1$. In the physical partitioning phase, all the processing nodes perform the physical partitioning of F_i's in parallel and create the grid files for all F_i's. Finally, the parallel grid file is built from F and the data is loaded into it.

4.1.1 Multidimensional Partitioning of F

The goal of this subsection is to present a dynamic programming formulation for the optimal solution of the logical and physical partitioning problems. For expository purposes, we present the model for the two dimensional case where each attribute has the same number of values. The model can be generalized to any number of dimensions and an arbitrary number of possible values per attribute. In the following discussion, we use the term *block* to denote physical or logical grid block.

Let $V_i = < v_{i1}, v_{i2}, ..., v_{in} >$ be the sorted set of values in D_i, the domain of attribute A_i, for $i = 1, 2$. For any pair of values v_{1i} and v_{2j}, there may be more than one record in F whose values on A_1 and A_2 are v_{1i} and v_{2j} respectively. Let FM be a $n \times n$ matrix, called *frequency matrix*, whose elements f_{ij}'s are the count of the records with values v_{1i} and v_{2j}. Thus,

$$f_{ij} = \|\{r \mid r \in F,\ r[A_1] = v_{1i} \text{ and } r[A_2] = v_{2j}\}\| \quad 1 \leq i, j \leq n,$$

where $r[A_i]$ denotes the value of record r on attribute A_i. In order to partition F into K blocks, we only need to partition the matrix FM. FM has to be partitioned by horizontal and vertical lines into K blocks such that some cost function is minimized. The partitioning of FM induces a similar partitioning on the records of F.

Given a pair of integers n_1 and n_2, they are called *permissible factors* of K if $n_1 \times n_2 = K$ and $n_i \leq n$ for $i = 1, 2$. We partition the matrix FM into K blocks by partitioning the rows into n_1 segments and the columns into n_2 segments. This is done by placing $n_1 - 1$ horizontal lines between the rows and $n_2 - 1$ vertical lines between the columns.

Let $FM(r, s)$ be the $n \times (s - r + 1)$ submatrix of FM which includes the elements f_{ij} for $1 \leq i \leq n$ and $r \leq j \leq s$, and π be a fixed partitioning of the rows of FM into n_1 segments. We denote by $OS_\pi(i, j)$ the total data skew or total data overflow from the partitioning of $FM(i, j)$ into n_1 blocks by horizontal lines in π only, i.e., no vertical lines are placed anywhere between columns i and j.

We consider possible partitions of the submatrix $FM(1, j)$ by placing vertical lines and without changing the position of the horizontal lines in π. Let $TOS_\pi(j, l)$ be the total cost incurred by optimal partitioning of $FM(1, j)$ using $l - 1$ vertical lines into $l \times n_1$ blocks given the fixed partitioning π. Based on the "principle of optimality" [HOR78], we can give the following recursive equation:

$$TOS_\pi(j, l) = MIN_{i=l-1,...,j-1}\{TOS_\pi(i, l-1) + OS_\pi(i+1, j)\} \text{ for } 1 < l \leq j$$
$$TOS_\pi(j, 1) = OS_\pi(1, j) \quad \text{for } j \geq 1.$$

The first term in the first equation is obtained by optimally partitioning the first i columns ($l - 1 \leq i \leq j - 1$) into $l - 1$ segments and placing a vertical line after column i, and the second part consists of columns $i + 1$ up to j, which are all placed in a single segment. The equation states that the optimal partitioning is obtained by a value of i which minimizes the sum of costs from these two parts. Thus, the optimal solution for the factors n_1 and n_2 under the fixed horizontal partitioning π is $TOS_\pi(n, n_2)$.

4.2 Model of Logical Partitioning

In the logical partitioning phase of the TOPLOADING algorithm, F is partitioned into $h \times v$ logical grid blocks. Letting $C = \frac{\|F\|}{h \times v}$ and replacing OS and TOS with $LSKEW$ and $LTSKEW$ (for logical skew parameters) in the recursive equation above, we get the mathematical model of the skew of the logical partitioning as follows.

$$LTSKEW_\pi(j, l) = MIN_{i=l-1,...,j-1}\{LTSKEW_\pi(i, l-1) + LSKEW_\pi(i+1, j)\}\ (1 < l \leq j)$$
$$LTSKEW_\pi(j, 1) = LSKEW_\pi(1, j) \quad \text{for } j \geq 1$$
$$LSKEW_\pi(i, j) = \sum_{k=1}^{h} \mid C_k - C \mid,$$

where, $C_1 = \sum_{y=i}^{j} \sum_{x=1}^{p_1} f_{xy}$, $C_h = \sum_{y=i}^{j} \sum_{x=p_{h-1}+1}^{n} f_{xy}$, $C_k = \sum_{y=i}^{j} \sum_{x=p_{k-1}+1}^{p_k} f_{xy}$ for $2 \leq k \leq h - 1$, and $p_1, ..., p_{h-1}$ are the positions of the $h - 1$ horizontal lines in π. The optimal solution of partitioning F for h and v, under a fixed horizontal partitioning π is $LTSKEW_\pi\ (n, v)$.

4.2.1 Model of Physical Partitioning

Let FM_k be the local frequency matrix of F_k for $0 \leq k \leq N - 1$. The physical grid partitioning uses processing node k to partition FM_k such that the total overflow of the physical grid blocks of F_k is minimized for $0 \leq k \leq N - 1$. Replacing OS_π and TOS_π

by $OVERF_\pi^k$ and $TOVERF_\pi^k$ in the recursive equation in subsection 4.1.1, we get the model of physical grid partitioning of F_k as follows for $0 \le k \le N-1$.

$$TOVERF_\pi^k(j,l) = MIN_{i=l-1,\ldots,j-1}\{TOVERF_\pi^k(i,l-1) + OVERF_\pi^k(i+1,j)\} \; (1 < l \le j)$$
$$TOVERF_\pi^k(j,1) = OVERF_\pi^k(1,j) \quad \textit{for } j \ge 1$$
$$OVERF_\pi^k(i,j) = \sum_{k=1}^{n_1} MAX(0, C_k - c),$$

where,
$C_1 = \sum_{y=i}^{j} \sum_{x=1}^{p_1} f_{xy}^k$, $C_{n_1} = \sum_{y=i}^{j} \sum_{x=p_{n_1-1}+1}^{n} f_{xy}^k$, $C_k = \sum_{y=i}^{j} \sum_{x=p_{k-1}+1}^{p_k} f_{xy}^k$ for $2 \le k \le n_1 - 1$, f_{xy}^k is a element of FM_k, and p_1, ..., p_{n_1-1} are the positions of the $n_1 - 1$ horizontal lines in π. The optimal solution of partitioning F_k for a pair of factors, x and y, under a fixed horizontal partitioning π is $TOVERF_\pi^k$ (n,y) for $0 \le k \le N-1$.

4.3 Performance Analysis of the TOPLOADING Algorithm

The *response time* of an algorithm is the elapsed time from the start to the end of the algorithm's execution. Theorem 4.1.1 gives the response time of the TOPLOADING algorithm.

Theorem 4.1.1. the response time of the TOPLOADING algorithm is

$$t_{i/o}(2P + \frac{3P}{N} + \frac{2P}{Nc} + \frac{h+v}{c}) + t_{cpu}(\frac{n^2hv}{N}\binom{n}{h} + \frac{2^{\frac{n}{2}}(nP)^{1.5}}{N} + \frac{|F|}{N} + N + h + v) + t_{comm}(n^2 + 3log_2N + \binom{n}{h} + \frac{(h+v)N}{log_2N} + B(1-\frac{1}{N})).$$

4.4 Random-Optimal Two Phase Algorithm

In this section we develop a more efficient random algorithm. It can give optimal or near optimal solutions for the data loading problem with high probability. It requires much lesser CPU, communication and I/O times than the algorithm in section 4.1. This is also a two phase algorithm. In the first phase, it uses random sampling to create the logical grid file structure for given file F and allocates the data of F to processing nodes according to the CMD function. In the second phase, it creates the physical grid file structure for F and loads F to the parallel grid file, using dynamic programming.

The logical partitioning in the first phase randomly selects samples from F, sorts them on each dimension, and determines partitioning points on each dimension using the sample quantiles as in [SSN92].

In the second phase, the physical partitioning is performed by using processing node i to solve a dynamic program on the subfile F_i, for $0 \le i \le N-1$. The mathematical model of the physical partitioning is the same as in section 4.1.3.

4.5 Performance Analysis of the ROTLADING Algorithm

Assuming that each sample needs one disk page access, Theorem 4.2.1 gives the response time of the ROTLOADING algorithm. In the following, SN is the sample size used in logical partitioning.

Theorem 4.2.1. The response time of the ROTLOADING algorithm is

$$t_{i/o}(SN + P + \frac{h+v}{c} + \frac{3P}{N} + \frac{2P}{Nc}) + t_{comm}(B(1-\frac{1}{N}) + log_2N) + t_{cpu}(\frac{2^{\frac{n}{2}}(nP)^{1.5}}{N} + 2SNlog_2SN + h + v + \frac{|F|}{N}).$$

4.6 Fully Random Two Phase Algorithm

For further reducing CPU cost, we propose a fully random algorithm. This algorithm also has two phases. In the first phase, it uses random sampling method to do the logical grid partitioning, creates the logical parallel grid file structure for a given file F, and allocates the logical grid blocks to processing nodes according to the CMD method. In the second phase, it uses processing node i to perform the physical partitioning of subfile F_i of F allocated to processing node i also by random sampling method, creates grid file structure for F_i, and loads F_i into the grid file for $0 \le i \le N-1$. Since this algorithm uses random sampling in both phases, we call it the fully random algorithm, denoted by FRLOADING.

4.7 Performance Analysis of the FR-LOADING Algorithm

Theorems 4.3.1 gives the response time of the FR-LOADING algorithm.

Theorem 4.3.1. The response time of the FR-LOADING algorithm is

$$t_{i/o}(SN_1 + P + \frac{P}{N} + \frac{2P}{N}(1 + \frac{1}{c}) + SN_2 + \frac{h+v}{c}) + t_{cpu}(2(SN_2 log_2 SN_2 + SN_1 log_2 SN_1) + h + v + \frac{|F|}{N} + \frac{n^2}{N}\sqrt{\frac{P}{N}}) + t_{comm}(B(1 - \frac{1}{N}) + log_2 N),$$

where, SN_1 and SN_2 are the sample sizes used in logical and physical partitionings.

5 Effect of Sample Size on Data Overflow

In all sampling based methods there is always a non-zero probability that the number of records assigned to a block will exceed page capacity and cause a page overflow. It is quite easy to deal with such overflows by creating a general overflow area or using separate chaining for each overflowed page. However, grid files do not allow data page overflows in order to guarantee at most two I/O's for exact match queries.

In this section we will derive a general bound on the probability that a block of the partitioning overflows. This can be used by the designer in determining proper values for s, the sample size, and α the storage utilization factor.

The idea is to use known bounds for a $d-1$ dimensional file to derive a bound for a d dimensional file. We derive it here only for $d = 2$. For ease of presentation we will use the term column to denote all blocks between a pair of successive vertical partitionings. Let $X = \{x_1, x_2, ..., x_M\}$ be a collection of M keys. Let $BX(s, \alpha, k, M)$ be a bound on the probability that in partitioning X into k parts based on a sample of size ks (using sample quantiles), one part contains more than $\frac{\alpha M}{k}$ keys. Let $XY = \{(x_1, y_1), (x_2, y_2), ..., (x_M, y_M)\}$ be a set of M two dimensional keys. We partition XY by taking a sample of size vs of the x values of each key and determine $v-1$ partitioning points along x-axis and then take a sample of size hs from the y values of each key. Let $BXY(s, \alpha, v, h)$ be a bound on the probability that a block of this partitioning has more than $\frac{\alpha M}{hv}$ elements of XY.

Theorem 5.1.

$$BXY(s, \alpha, v, h) \leq v \times [BX(s, \alpha, v, M) + (1 - BX(s, \alpha, v, M)) \times h \times BX(s, \alpha, h, \frac{M}{v})].$$

Proof (Sketch): The first term in the angular brackets represents the fact that in case one column of the partitioning overflows at least one block in that column will also overflow. The second term represents a bound on the probability of the event that a column does not overflow (i.e. has at most $\alpha M/v$ keys but at least one block in that column still overflows.

The above formula is general for any known one dimensional bound. In particular we use the best known bound given in [SSN92,SES92]

$$BX(s, \alpha, k, M) = \alpha^s \left(\frac{k-\alpha}{k-1}\right)^{ks-s}$$

to derive:

$$BXY(s, \alpha, v, h) \leq v\alpha^s [\left(\frac{v-\alpha}{v-1}\right)^{vs-s} \left(1 - h\alpha^s \left(\frac{h-\alpha}{h-1}\right)^{hs-s}\right) + h\left(\frac{h-\alpha}{h-1}\right)^{hs-s}]$$

Figure 1 shows the quality of this bound for α between 1.5 and 1.8 and $h = v = 50$.

6 Simulation and Comparisons

In this section, we examine the performance of the proposed algorithms by simulation. We investigate the effect of sample size on the total data skew of logical or physical grid blocks resulting from sampling partitioning methods. We also compare the response time of the algorithms using the analytical results in section 4. The system parameters used in our simulation are as follows:

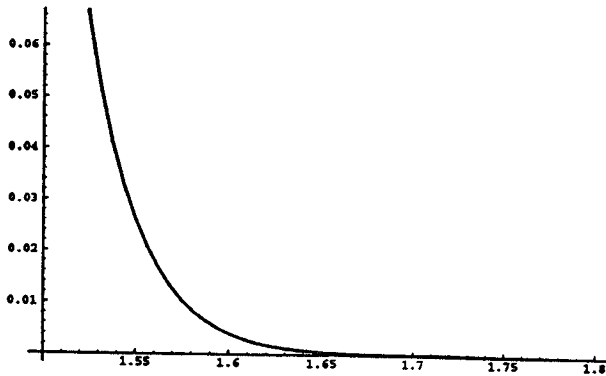


Figure 1: Bounds on the probability of at least one block overflowing.

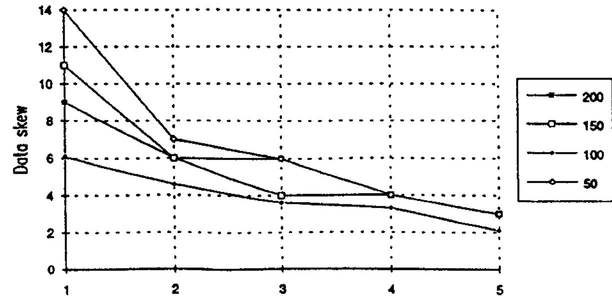


Figure 2: Effect of sample size on data skew for different file size.

Number of processing nodes:	$N = 64$.
CPU processing rate:	20MIPS.
I/O bandwidth:	5Mbytes/second.
Effective communication channel bandwidth:	10Mbytes/second.
Capacity of disk page:	4k bytes.

These parameters are comparable with that in [OMI92] and [HUA90].

6.1 Effect of Sample Size on Data Skew of Logical Partitioning

We ran the algorithm on 4 randomly generated files with different size each. The sizes of the files are 50000, 100000, 150000 and 200000 records. The block size is 100 records. Fig. 2 shows the results. In the legend of the figure, 50, 100, 150 and 200 are in units of 1000. From Fig. 2, we can see that the sampling partition method works well for a large range of files. When the sample size is greater than 4%, the "data skew" is less than 4% and the method is almost independent of file size.

6.2 Effect of Sample Size on Data Overflow of Physical Partitioning

To examine the effect of sample size on total overflow of physical grid blocks, we implemented the physical sampling partition method in the FRLOADING algorithm. We ran the program on a file of 1000000 randomly generated records for grid block sizes of 400 records with system page capacity 400. The domain size of each attribute of the file is 1000000.

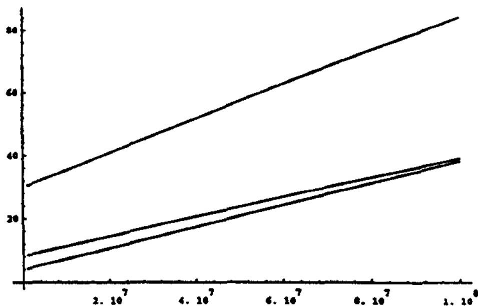
We also implemented the dynamic programming physical partitioning algorithm. We ran the sampling and dynamic algorithms on a file of 10000 randomly generated records with system page capacity 100. The domain size of each attribute of the file is 15. In Table 3, we compare A.O. and N.O. between the dynamic programming algorithm and the sampling algorithm for various values of α. The sample size is 3% of the total file size. As we can see from this table sampling provides a reasonable estimation as compared with the optimal solution.

Methods		$\alpha = 1$	$\alpha = 1.1$	$\alpha = 1.3$	$\alpha = 1.5$	$\alpha = 1.7$	$\alpha = 1.9$
Dynamic program	A.O.	32	36	18	11	7	6
	N.O.	16	10	9	7	4	0
Sampling	A.O.	49	46	31	23	15	7
	N.O.	19	15	13	10	5	2

Figure 3: **Table 3. Comparisons of A.O. and N.O between dynamic programming and sampling algorithms.**

6.3 Comparison of Response Time

Fixing n, the size of domains of all attributes, to 65, we first compare the response time of the algorithms for increasing file size using the analytical results of section 4. For comparison, we assume that record



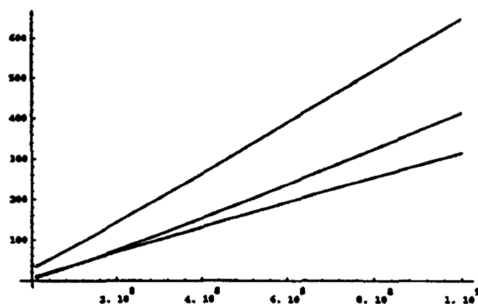


Figure 4: Comparison of response time for varying file size.

size is 200 bytes, $h = v = 64$, and sample size is 5000. Fig. 4 shows the comparison of response times for files with sizes varying from 10^5 bytes to 10^9 bytes. In the two figures, the time unit is seconds. It is obvious that all the algorithms are very efficient for a large range of file sizes. In these figures, i.e. for $n = 65$, the TOPLOADING algorithm is always slower than the other two because of the high cost of dynamic programming. However, this algorithm can provide an optimal solution and can work very efficiently for small n. Fig. 4 also shows that the ROTLOADING algorithm is faster than the FRLOADING algorithm for files with size less than 10^8 bytes. The reason is that the response time in this case is dominated by the I/O time for accessing samples and the FRLOADING algorithm needs more time to read the samples in the physical partition phase than the ROTLOADING algorithm. In the following discussion, we fix the file size to 10^8 bytes. We assume that record size is 200 bytes, $h = v = 64$, and sample size is 5000.

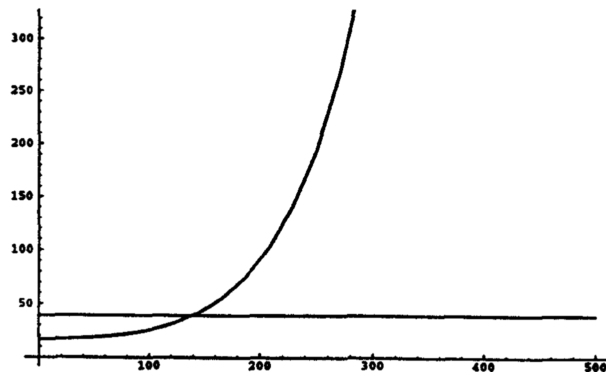


Figure 5: Comparison of response time of the ROTLOADING and FRLOADING algorithms for varying size of attribute domains

Fig. 5 shows the comparison of the response time of the ROTLOADING and FRLOADING algorithms when n varies from 10 to 500. This figure shows that the ROTLOADING algorithm is more sensitive to n than the FRLOADING algorithm due to the dynamic programming in the physical partitioning. When n is less than 150, the ROTLOADING algorithm is more efficient than the FRLOADING algorithm. The reason is that the I/O time for accessing samples is the main part of the response time in this case and the FRLOADING algorithm needs more I/O time to read the samples in the physical partition phase than the ROTLOADING algorithm.

7 Conclusions and Future Research

Three algorithms for loading parallel grid files, namely TOPLOADING, ROTLOADING and FRLOADING, have been proposed in this paper. The TOPLOADING algorithm is based on dynamic programming. It gives optimal solutions for the logical and physical partitioning problems. However it is only efficient for small n, the number of different values in domains of attributes of a given file. The ROTLOADING algorithm is based on the combination of the dynamic programming and estimation of population quantiles using sampling. It gives a near optimal solution for the logical partitioning problem by sampling and optimal solution for physical par-

titioning problem by dynamic programming. It is less sensitive to n and much more efficient than the TOPLOADING algorithm. However, it will be inefficient when n is very large. The FRLODING algorithm is the most efficient algorithm. It gives near optimal solutions for logical and physical partitioning problems by sampling. The theoretical and simulation results show that this algorithm can efficiently work for any n and give very near optimal solutions with small number of samples.

Our ongoing research is addressing the following issues:

1. Analysis and experimentation of our algorithms for different data distributions including correlations between attribute values on the two dimensions.

2. Developing algorithms for loading other data structures such as Quad tree, R-tree, etc.

3. Developing loading algorithm for other parallel computing architectures.

References

[GAK90] Abdel Ghaffar K. and El Abbadi A. On the Optimality of disk allocation for cartesian product files. In *proceedings of ACM Symposium on Principles of Database Systems*, (April 1990), 258-264.

[FKK91] Fujwara T., Ito M, Kasami t., Katakoa M. and Okui J., Performance analysis of disk allocation method using error-correcting codes. IEEE Transactions on Information Theory 37, 2(1991),379-384

[HUA90] Kien A. Hua and Chiang Lee, An Adaptive Data Placement Scheme for Parallel Database Computer Systems, in *Proc. of Inter. Conf. on Very Large Data Bases (VLDB)*, Australia, 1990.

[HOR78] T.H. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, *Computer Science Press*, 1978.

[LSR92] Jianzhong Li, Jaideep Srivastava and Doron Rotem, CMD: An Multidimensional Declustering Method for Parallel Database Systems, in *Proc. of Inter. Conf. on Very Large Data Bases (VLDB)*, Canada, Aug. 1992.

[NIC92] T. M. Niccum, J. Srivastava, and Jianzhong Li, A Hash Based Parallel Join for Relations with Coordinate Modulo Declustering (CMD) Storage, Submitted for publication, 1992.

[NIE84] J. Neievergelt, H. Hinterberger and K.C. Sevcik, The Grid File: An Adaptable, Symmetric Multikey File Structure,

[OMI92] E. Omiecinski and E. T. Lin, The Adaptive-Hash Join Algorithm for a Hypercube Multicomputer, *IEEE Trans. on Parallel and Distributed Systems*, Vol.3, No.3, May, 1992. *ACM Trans. on Database Systems*, Vol. 9, No. 1, 1984.

[ROS81] Arnold L. Rosenberg and Lawrence Snyder, Time- and Space-Optimality in B-Tree, *ACM Transaction on Database Systems*, Vol.6, No.1, March 1981.

[ROT88] Doron Rotem and Arie Segev, Algorithms for Multidimensional Partitioning of Static Files, *IEEE Tran. on Software Engineering*, Vol. 14, No. 11, Nov. 1988.

[SES92] S. Seshadri, Probabilistic Methods in Query Processing, *Ph.D. thesis*, Department of Computer Science, University of Wisconsin-Madison, 1992.

[SSN92] S. Seshadri and J. F. Naughton, Sampling Issues in Parallel Database systems, in *Proceedings of the 3rd International Conference on Extending Database Technology*, Vienna, Austria, March 1992.

[STO86] M. Stonebraker, The Case for Shared Nothing, *Database Engineering*, 9(1), 1986.