Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Parallel Computing 29 (2003) 1419–1444

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Parallel bulk-loading of spatial data

Apostolos Papadopoulos, Yannis Manolopoulos *

Department of Informatics, Aristotle University of Thessaloniki, 54006 Thessaloniki, Greece

Received 8 April 2002; accepted 12 May 2003

Abstract

Spatial database systems have been introduced in order to support non-traditional data types and more complex queries. Although bulk-loading techniques for access methods have been studied in the spatial database literature, parallel bulk-loading has not been addressed in a parallel spatial database context. Therefore, we study the problem of parallel bulk-loading, assuming that an R-tree like access method need to be constructed, from a spatial relation that is distributed to a number of processors. Analytical cost models and experimental evaluation based on real-life and synthetic datasets demonstrate that the index construction time can be reduced considerably by exploiting parallelism. I/O costs, CPU time and communication costs are taken into consideration in order to investigate the efficiency of the proposed algorithm.

Keywords: Parallel databases; Spatial access methods; Bulk-loading; Query processing

1. Introduction

Parallel DBMSs have been developed as research prototypes [9,31] and complete commercial systems. The benefits of these systems can be easily understood taking into consideration the large computational power and the huge amounts of data that modern applications require. Another research area with major interest is supporting space in database systems (spatial database systems) [15], where objects are not single-valued and range from points in a multidimensional space to complex polygons with holes. In order to efficiently support applications requiring non-conventional data, a number of spatial access methods have been proposed. The most important characteristic of these methods is their efficiency in answering user queries involving

* Corresponding author. Tel.: +30-31-996363; fax: +30-31-996360.
E-mail address: manolopo@csd.auth.gr (Y. Manolopoulos).

0167-8191/$ - see front matter
doi:10.1016/j.parco.2003.05.003

spatial relationships between the objects [2,16,24,29]. An excellent survey on multidimensional index structures has been reported in [13].

Basically, there are two approaches that can be followed in order to build a spatial access method. The first technique involves individual insertions of the spatial objects, meaning that the access method must be equipped to handle insertions. The second technique involves building the access method by using knowledge of the underlying dataset. Evidently, the second technique requires that the data are available in advance. However, this situation occurs quite frequently even in dynamic environments. For example, data can be archived for many days in data warehouses and in order to answer queries efficiently, access methods must be constructed first. Several clustering methods and other data mining tasks require the presence of an index structure. Moreover, even in conventional relational database systems, a user may create an index on demand on a specific attribute of a relation (e.g. using SQL on can pose: CREATE INDEX SALIDX ON TABLE EMPS (salary)) in order to speed-up join queries based on this attribute. These observations have triggered researchers in the database community to investigate bulk-loading techniques towards fast access method generation. The benefits of bulk-loading are summarized below:

- the access method can be constructed faster than by using individual insertions,
- the quality of the produced access method can be optimized since the underlying data are known in advance,
- the space utilization is usually much better (in some cases approaches 100%) and therefore less disk operations are required to answer a query.

Although the literature is rich in bulk-loading techniques for spatial access methods, to the best of the authors' knowledge the problem of parallel bulk- loading in spatial database systems has not been given much attention. The challenge is to exploit parallelism in order to fulfill both efficient index generation and high quality of the produced index. Therefore, we study parallel techniques for bulk-loading, where we assume that the environment is composed of a number of processors based on a shared-nothing architecture, where each processor manages its own disk(s) and main memory. The method is studied for both the efficiency during index generation and the quality of the produced index. Moreover, the speed-up, size-up and scale-up of the method is studied and several experimental results are demonstrated based on real-life and synthetic datasets.

During our study we assume that no reorganization of the data takes place. In other words, after the completion of the index construction process, the data remain assigned to the same processor. However, in order to guarantee load balance during index construction, it is necessary for some processors to transmit the spatial information of the objects to other processors, without transmitting the whole record or the objects' detailed geometry.

The rest of the paper is organized as follows. The next section presents the appropriate background and related work on sequential bulk-loading techniques. Section 3 presents the phases of parallel bulk-loading and provides a cost model to estimate the cost of the parallel bulk-loading algorithm. In Section 4 results are given based

on experimental performance evaluation. Finally, Section 5 concludes the paper and motivates for further research.

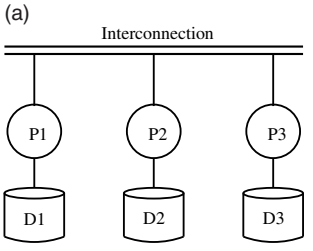
2. Background and related work

The architecture of a parallel spatial database system is depicted in Fig. 1(a). Each processor manages its own memory and disk(s), and interprocessor communication is achieved by using messages over the interconnection. No assumptions are posed with respect to the nature of the interconnection. Therefore, the algorithms can be applied in loosely or tightly coupled multiprocessors.

Let SR be a spatial relation, with a spatial attribute sa and several alphanumeric attributes. Each processor contains a portion of SR. The distribution of the data to the processors may follow range partitioning, hash partitioning or any other declustering approach [11]. Moreover, one or more attributes may participate in the partitioning process. No assumptions are made for the partitioning method that has been used to decluster the relation. As an example, Fig. 1(b) depicts the case where the relation *CITIES* has been partitioned using range partitioning on the *population* attribute. The spatial attribute *location* corresponds to points in the 2-D Euclidean space. Each location may be the map coordinates of the corresponding city. Without loss of generality, we will assume that the spatial relation has only one spatial attribute. The case for multiple spatial attributes can be handled in a similar manner, by applying the method for each spatial attribute. The problem that is investigated is stated as follows:

Problem definition. Given a spatial relation SR with one spatial attribute sa, which is declustered across a number of processors, determine an efficient way to construct a spatial access method with respect to the spatial attribute sa, by exploiting parallelism.

The process of constructing an index *from scratch* is also known as bulk-loading. There are three methods that can be applied in order to construct a multidimensional access method for a dataset:



(b) CITIES

Name	Population	Location	
Athens	4,000,000	400, 400	Processor P1
Thessaloniki	1,000,000	500, 1000	
Larissa	300,000	300, 700	Processor P2
Patras	200,000	100, 400	
Heraklion	150,000	100, 500	Processor P3
Kavala	100,000	600, 1000	
Drama	50,000	700, 1000	

Fig. 1. (a) Architecture of a parallel database system with three processors and (b) range partitioning of relation *CITIES* with respect to attribute *population*, using three processors.

- *Individual insertions*. The dataset is scanned and each object is inserted into the access method by exploiting the specific insertion algorithm.
- *Bottom-up*. The dataset is sorted with respect to a criterion that preserves spatial proximity, the leaf level of the access method is formulated and finally the upper tree levels are constructed by the same procedure.
- *Top-down*. The dataset is partitioned according to a splitting decision policy, and each partition is further repartitioned until the minimum node capacity requirement is fulfilled.

The problem has been studied in the literature and a number of very promising methods have been proposed. In [27] the authors propose a packing algorithm for the R-tree access method. This method was later refined by Kamel and Faloutsos in [18], where the Hilbert value of each object is used, and then a total order of the objects is performed. Another approach to bulk-loading R-trees is proposed in [21]. The dataset is partitioned in chunks and a number of R-tree nodes is created for each chunk. An algorithm for grid-file bulk-loading has been presented in [22]. The authors in [8] illustrate a way to bulk-load an access method that is based on metric distances (called the M-tree). A generic approach in bulk-loading has been proposed in [3]. The authors propose a method that can be applied to the majority of the access methods, and considerably speeds-up the index construction process. However, the construction process does not take into account all the available data. The index is constructed faster than the one-at-a-time approach, but no improvements on the quality and space utilization are achieved according to the authors. An improvement on this idea has been reported in [1]. Another approach for bulk-loading is proposed in [4] where the authors introduce the concept of non-balanced splitting, in order to guarantee the efficiency of the resulting access method in multidimensional query processing. Finally, in [5] generic bulk-loading techniques are categorized and evaluated.

The main common characteristic of the aforementioned approaches is that a uniprocessor system is assumed. Given that parallel database systems is one of the research directions towards efficient query processing, it is important to provide parallel bulk-loading operations for spatial databases.

3. Parallel bulk-loading

Although the problem of index bulk-loading has been studied by many researchers, the parallel version of the problem did not receive the required attention. Parallel database systems are currently operational in large companies and organizations, and there is a lot of activity in the area, towards improving the performance of modern demanding applications. Efficient index construction techniques need to be developed, in addition to efficient query processing engines, exploiting the potentials of the system (e.g., many processors, many disks). In the case of a simple 1-D index, such as the B^+-tree, the problem of parallel bulk-loading can be solved by applying efficient parallel external sorting algorithms [10,34]. However, there is no work

addressing this problem for higher dimensionalities, to the best of the authors' knowledge. The following two realistic assumptions are introduced, in order to attack the problem:

- the spatial relation is horizontally fragmented across all or a subset of the processors,
- the number of processors in the system is a power of two. [1]

As mentioned in the previous section, the spatial relation may be partitioned according to one or more attributes. Therefore, we can not guarantee that objects that are close in the spatial domain are stored on the same processor. For example, if the records are partitioned according to a hash function on a non-spatial attribute, two objects close in space are likely to be positioned in different processors. The same applies to other partitioning techniques like range partitioning. The benefits of constructing a spatial index in such an environment is twofold.

(1) Efficient query processing techniques can be applied, since the sequential scanning of the data in each processor is avoided. Consequently, range queries, nearest-neighbor queries and spatial join queries can be answered more efficiently.
(2) The constructed index can serve as a yardstick towards redistribution of the records based on the spatial attribute. For example, in [20] the authors propose a declustering technique in order to distribute the leaf level of an R-tree across a number of workstations. In order to apply such a page distribution scheme, at least the leaf level of the index must be available.

3.1. Methodology

We decompose the parallel bulk-loading procedure into a number of phases. Then we illustrate techniques in order to implement each one of these steps. Table 1 contains the symbols used for the rest of the study.

3.1.1. Random sampling

One of the processors is selected as the coordinator. Each of the other processors selects S_j records from its local data by using random sampling. Random sampling assumes that each object has the same probability to be selected from a population of N_j objects. Then, the spatial attribute of these S_j records is transmitted to the coordinator. Evidently, the sample size plays an important role for the quality of the produced index. If the sample size is too small, the data distribution is not reflected properly. On the other hand, a large sample size will cause performance degradation due to interconnection and I/O overheads. This trade-off will be investigated in detail during the performance evaluation.

[1] Later we discuss how this assumption can be relaxed.

Table 1
Symbols and definitions

Symbol	Definition
f	Sampling factor, $f \in (0,1]$
s	Average number of samples per processor
S_j	Sample size of jth processor
$\mathbb{S}_j$	Sample set of the jth processor
D	Data space dimensionality
N	Total number of objects (records of the spatial relation)
N_j	Number of objects hosted in processor P_j
P	Number of processors in the system
P_j	jth processor
R_i	ith region
$\lvert R_i \rvert$	Number of sampled objects in partition R_i
$\lVert R_i \rVert$	Total number of objects in partition R_i
$O_{\mathfrak{p}}$	Total number of objects preserved to the same processor
$O_{\mathfrak{t}}$	Total number of objects transmitted to other processors
k_{ij}	Number of objects from the sample set of processor P_j contained in region R_i
n_{ij}	Total number of objects from processor P_j contained in region R_i

In this study we assume that the sampling factor f is the same for all processors. Therefore, processor P_j collects $S_j = N_j \cdot f$ records from its local data. Using this technique, the more records a processor contains, the more data are contained in the sample set of this processor. This is needed because some processors may contain more data than the others (data skew), and therefore more samples are needed in order to capture the distribution of the spatial attribute. In our investigation we also use the parameter s which is the average number of samples produced by each processor. If initially the processors contain the same number of data objects, then $S_j = s \ \forall j$. The sampling factor f and the parameter s are related with the following equation: $f \cdot N = P \cdot s$.

3.1.2. Space partitioning and region assignment

The first action that should be performed is to create a suitable partitioning of the data space. Then, each partition of the space will be assigned to a single processor. The key issue here is to exploit the sampled data that have been collected in the previous phase.

The collected spatial information is used to construct the partitioning of the space, using space decomposition. The output of the decomposition process is P regions of the space. The decomposition is performed in a way similar to the construction of a *kd*-tree (Fig. 2).

The next issue is to assign each of the regions to a processor. Each partition contains objects from one or more processors. Therefore, the partition R_i can be described as a vector $V_i = (k_{i1}, k_{i2}, \ldots, k_{iP})$ where k_{ij} is the number of objects from the sample of processor P_j that are contained in the current partition. The challenge is to assign regions to processors in such a way that the number of objects preserved to the same processor is maximal. Fig. 3 illustrates two possible assignments. For

(a)

```
Algorithm spacePartitioning(int nb_regions)
1. begin
2.   int regions;
3.   regions = 0;a4o&o4a

4.   while (regions < nb_regions)
5.   begin
6.     split_axis = splitAxis();
7.     split_pos = splitPos(split_axis);
8.     performSplit();
9.     regions++;
10.  end;
11. end;
```

(b)

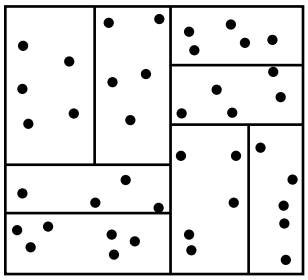


Fig. 2. (a) Space partitioning algorithm and (b) example of a 2-D space decomposition with eight regions.

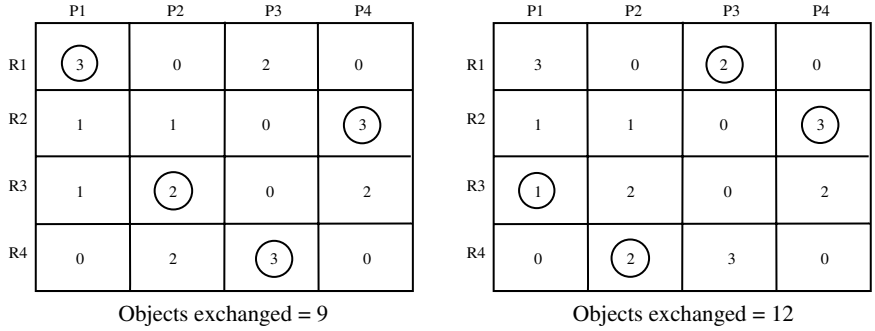


Fig. 3. Two possible assignments of four regions to four processors. The assignment on the left is optimal with respect to the number of transmitted objects.

each assignment the number of objects exchanged among the processors is illustrated. Columns represent the processors and rows represent the data space regions. The values in the cells of the matrix are the n_{ij} values. A circle in the position (R_i, P_j) of the matrix denotes that partition R_i has been assigned to processor P_j.

Evidently, there are $P!$ possible assignments of regions to processors. If P is small, the best assignment can be determined using exhaustive search. However, if P is large which is very common in massively parallel computers (hundreds of processors) a more efficient assignment algorithm is needed. By observing the problem it is not difficult to realize that it is equivalent to the weighted bipartite graph matching problem [7]. The graph bipartitions are the data space regions (vertex set $\mathbb{R}$) and the processors (vertex set $\mathbb{P}$). The weight of each edge connecting a data space region R_i to a processor P_j declares the number of objects from P_j that are enclosed by region R_i. Therefore, the cost of the matching is equivalent to the total number of objects that are preserved to the same processor. Evidently, the number of preserved objects O_p and the number of transmitted objects O_t are related by: $O_p = N - O_t$.

Alternatively, a more simple greedy algorithm can be used in order to avoid the high complexity of the bipartite matching algorithm. Each space region is assigned

to the processor that dominates in the corresponding region. A processor P_j dominates in a region R_i if the majority of the sample points contained in R_i belong to processor P_j. The regions are sorted according to the domination information. Then, the sorted regions are checked one-by-one and each region is assigned to a processor. If a processor has already received a region it is not considered any more. Although this algorithm does not produce always the optimal solution, it is a good alternative which requires less computation overhead.

3.1.3. Load balancing

The next target is to assign approximately the same number of objects to each processor, in order to guarantee that each one receives the same amount of work to perform.

Fig. 4 illustrates the load balancing algorithm that is executed in each processor. The objective is twofold: (i) to send the spatial attribute of the objects that have been assigned to other processors and (ii) to receive the spatial attribute from other processors. Evidently, in the ideal case, no data exchange is needed. However, in the general case several objects must be send from one processor to another. We note that only the value of the spatial attribute of each object is needed and not the entire record.

The issue that need to be investigated at this point, is if load balancing is achieved. In other words, does the partitioning phase guarantee that each processor receives approximately the same amount of work? The following propositions answer this question positively.

Proposition 1. *If the sampling factor f is sufficiently large, then each processor will receive approximately the same number of objects after the load balancing phase.*

Proof. "f is sufficiently large" means that the number of produced samples is adequate to capture the distribution of the data objects. Recall that the partitioning phase partitions the sample set in P regions each containing $f \cdot N/P$ sample points. Let k_{ij} denote the number of sample points from P_j contained in Ri and n_{ij} denote the total number of data points from P_j contained in R_i. Therefore

```
Algorithm loadBalance
1.    begin
2.        int i;
3.        receive partitioning information from coordinator;
4.        for i=1 to P do
5.            for each object o_x in R_i do
6.                determine processor P(o_x) that o_x must be sent to;
7.                if P(o_x) == P_j keep the object;
8.                else send o_x to P(o_x);
9.            endfor
10.       endfor
11.       receive objects from other processors;
12.   end
```

Fig. 4. The *loadBalance* method executed in each processor.

$$|R_i| = \sum_{j=0}^{P-1} k_{ij} = f \cdot \frac{N}{P}$$

The total number of points in a region R_i is given by

$$\|R_i\| = \sum_{j=0}^{P-1} n_{ij}$$

If the data distribution is well described by the produced sample set, then the number of objects from P_j and the number of sample objects from P_j are related with the following equation (evidently the data distribution is best described by setting $f = 1$, i.e., all data objects are selected as samples):

$$k_{ij} = n_{ij} \cdot f$$

By combining the above equations we obtain

$$\|R_i\| = \sum_{j=0}^{P-1} \frac{k_{ij}}{f} = \frac{1}{f} \cdot f \cdot \frac{N}{P} = \frac{N}{P} \qquad \square \tag{1}$$

The above proposition can be used in order to define a simple model regarding the number of objects assigned to a processor after the completion of the load balancing phase. However, this model cannot be used to determine the probability that a region assigned to a processor may contain many more objects than the average. Therefore, we continue by providing a more detailed model which is based on similar analytical considerations regarding parallel sorting [6,30] and it is adapted for multidimensional datasets.

Let E_i be the event that after the load balancing phase, region R_i contains more than $r_{\text{skew}} \cdot N/P$ data objects, where $r_{\text{skew}} \geqslant 1$. Intuitively the factor r_{skew} represents the deviation of the number of objects in a region from the average value N/P, which is the ideal case. We are interested in determining the probability that at least one region will contain more than $r_{\text{skew}} \cdot N/P$ data objects after the completion of the load-balancing phase. Let R_i be an arbitrary region of the space which contains exactly $r_{\text{skew}} \cdot N/P$ data objects. We search the probability that from a total of $P \cdot s$ samples generated by all processors fewer than s samples exist in R_i, where s satisfies the equation: $P \cdot s = f \cdot N$. The space partitioning method assigns exactly s sample points to each region R_i. Therefore, if less than s sample points are required to guarantee that exactly $r_{\text{skew}} \cdot N/P$ data points are in region R_i, R_i will contain more than $r_{\text{skew}} \cdot N/P$ data points.

Proposition 2. *Let R_i be an arbitrary region of the space containing exactly $r_{\text{skew}} \cdot N/P$ data points. Let also X be a random variable representing the number of sample points contained in R_i. Then, the probability* $\text{Prob}[X < s]$ *that less than s sample points are contained in R_i is given by*

$$\text{Prob}[X < s] = \sum_{x=0}^{s-1} \frac{\binom{r_{\text{skew}} \cdot N/P}{x} \cdot \binom{N - r_{\text{skew}} \cdot N/P}{P \cdot s - x}}{\binom{N}{P \cdot s}}$$

Proof. Assume that the $r_{\text{skew}} \cdot N/P$ objects of region R_i are painted "red" and the rest $N - r_{\text{skew}} \cdot N/P$ objects are painted "blue". if $P \cdot s$ points are selected randomly from the N data points without replacement, the probability $\text{Prob}[X = x]$ that exactly x out of $P \cdot s$ sample points are colored "red" (i.e. they belong in region R_i) is given by the hypergeometric distribution as follows:

$$\text{Prob}[X = x] = \frac{\binom{r_{\text{skew}} \cdot N/P}{x} \cdot \binom{N - r_{\text{skew}} \cdot N/P}{P \cdot s - x}}{\binom{N}{P \cdot s}}$$

Therefore the probability that less than s sample points are contained in R_i can be calculated as follows:

$$\text{Prob}[E_i] = \text{Prob}[X < s] = \sum_{x=0}^{s-1} \text{Prob}[X = x] \qquad \square$$

Let E the event that at least one region contains more than $r_{\text{skew}} \cdot N/P$ data points. Obviously $E = E_0 \cup E_2 \cup \cdots \cup E_{P-1}$. The events E_i for $i = 0, \ldots, P-1$ are not mutually exclusive. Therefore the sum of their individual probabilities is greater than the probability of their union

$$\text{Prob}[E] < \sum_{i=0}^{P-1} \text{Prob}[E_i]$$

In [30] an upper bound is derived for the above expression

$$\text{Prob}[E] \leqslant P \cdot r_{\text{skew}}^{s} \cdot \left(\frac{P - r_{\text{skew}}}{P - 1} \right)^{(P-1)\cdot s}, \quad s = \frac{f \cdot N}{P} \tag{2}$$

Fig. 5 depicts a graphical representation of Eq. (2) and illustrates the relationship among the number of processors, the number of samples per processor and the skew ratio r_{skew}. The probability of the event that at least one region contains more than $r_{\text{skew}} \cdot N/P$ data points is set to 0.999, and the total number of data points N is 1,000,000. It is shown that for constant skew ratio more samples are needed as the number of processors increases.

3.1.4. Local index construction

After the completion of the load-balancing step, each processor contains a collection of triplets of the form (*recordID*, *processorID*, *point*), where *recordID* is a record identifier, *processorID* is the processor where the record details are stored and *point* is the value of the spatial attribute of the record. The next step is to build a local (par-

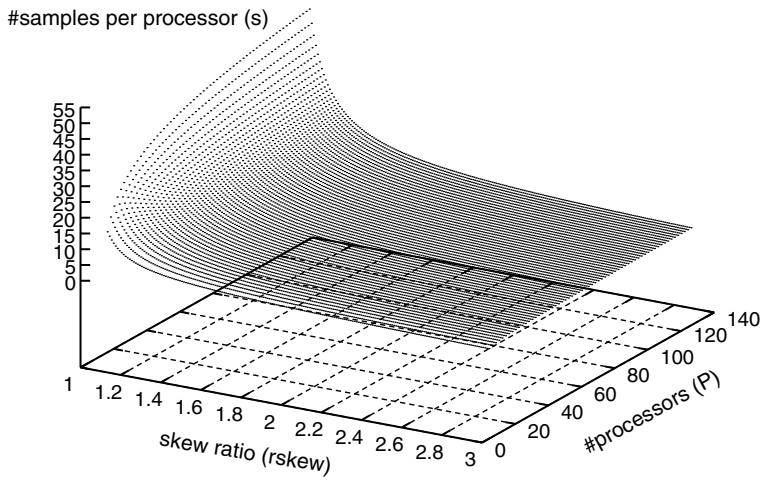


Fig. 5. Relationship among the number of processors, the number of samples per processor and the skew ratio.

tial) index for these triplets, based on the values of spatial attribute. As we have already mentioned in a previous subsection, several techniques have been proposed for building an R-tree like spatial access method in a uniprocessor system, using bulk-loading. Any one of these techniques can be applied to construct the local spatial index of each processor.

3.1.5. Global index composition

The final step that completes the parallel bulk-loading algorithm is the composition of the global spatial index. Each processor submits the upper levels of its local index to the coordinator. The coordinator receives this information and composes a global index, by constructing a common root. Each of the local indexes becomes a subtree of the new root. A potential problem that may arise is that the heights of the local trees may be different. This problem as well as several alternatives are discussed later.

3.2. Cost model

The existence of a cost model aids the query optimizer in selecting an appropriate query execution plan (QEP). Often, during query execution, indexes need to be constructed "*on-the-fly*". For example, consider a query that requires a join between two subsets of two relations R and S. Two different query execution plans are illustrated in Fig. 6.

In the first QEP two selection operations are performed and then the nested-loop join algorithm is used to perform the join. In the second QEP, each selection operation is followed by a bulk-loading operation which constructs a temporary index for each subset. Finally, the join is performed by exploited the constructed indexes. In

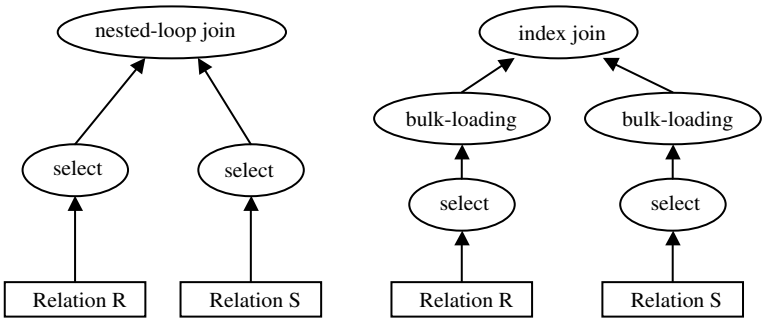


Fig. 6. Two different query execution plans.

order to determine the most promising QEP regarding the query execution time, accurate cost models are required, at least for the estimation of the major costs involved. Towards this direction, in the remaining of this section we provide analytical expressions for the major costs of the parallel bulk-loading algorithm.

A cost is assigned to every major step of the parallel bulk-loading algorithm. This cost measures the estimated elapsed time (in seconds) for the completion of the corresponding phase. Although in some cases this cost is very small, it is included for completeness. Table 2 illustrates the cost-model parameters. The values of the parameters T_{comp}, T_{move}, T_{swap} and T_{sample} have been determined on a 2.4 GHz Intel Pentium IV machine running Windows 2000.

The spatial relation is composed of records. Each record is composed of the record identifier (S_{number} bytes), the value of the spatial attribute ($D \cdot S_{\text{number}}$ bytes) and the rest irrelevant attribute values with total size 128 bytes. Therefore each record has a total size of: $S_{\text{rec}} = (D+1) \cdot S_{\text{number}} + 128$ bytes.

The cluster size S_{cluster} denotes the maximum number of disk pages that can be fetched in a single I/O operation. Therefore, the cost of reading 4 consecutive pages costs one random access plus the number of retrieved bytes divided by the disk transfer rate. If x clusters must be read, the corresponding time is:

$$T_{\text{i/o}}(x) = x \cdot (T_{\text{cluster}} + T_{\text{cpu-io}}) \tag{3}$$

The cost of transmitting x bytes through the interconnection can be approximated by the following equation:

$$T_{\text{trans}}(x) = \frac{x + S_{\text{header}} \cdot x/S_{\text{msg}}}{B} + \left\lceil \frac{x}{S_{\text{msg}}} \right\rceil \cdot T_{\text{msg}} \tag{4}$$

This costs includes the transmission cost of the physical media and the overhead incurred per message. Based on these two fundamental costs, next we give the cost for each of the phases of the parallel bulk-loading algorithm.

Random sampling (RS). The cost of the random sampling phase is dominated by the processor that has to produce the largest sample set. Evidently, this is the proces-

Table 2
Parameters of the cost model

Parameter	Definition	Default value
B	Interconnection bandwidth	100 MBit/s
T_{page}	Time spent on the disk for a ransom page I/O	0.01 s
T_{cluster}	Time spent on the disk for a ransom cluster I/O	0.01 s
T_{msg}	Time spent on CPU to compose a message	0.001 s
T_{comp}	Time spent on CPU to perform number comparison	2.5×10^{-9} s
T_{move}	Time spent on CPU to move a number to another memory location	2.6×10^{-9} s
T_{swap}	Time spent on CPU to swap the contents of two memory locations	1.9×10^{-8} s
T_{sample}	Time spent on CPU to produce a random sample	2.6×10^{-8} s
$T_{\text{cpu-io}}$	CPU overhead for each I/O	0.0001 s
S_{number}	Size of a number	4 bytes
S_{page}	Size of a disk page	4096 bytes
S_{cluster}	Cluster size	4 pages
S_{mem}	Size of a processor memory	100 pages
S_{msg}	Size of a message for data transmission (IEEE 802.3)	1500 bytes
S_{header}	Size of the message header for IEEE 802.3 protocol	72 bytes
S_{rec}	Size of a record	128 bytes
r_{skew}	Skew ratio ($\max N_j/\text{avg}N_j$, $j=1,\ldots,P$)	
$r_{\max}$	$\max N_j/N$, $j=1,\ldots,P$	

If the value of a parameter is not given, the default value is assumed.

sor which contains $max\{N_j\}$ $(j=1,\ldots,P)$ number of records. Let P_k be this processor. Therefore, P_k produces $N_k \cdot f$ samples. Each sample costs one disk access plus the time spent on the CPU. The total sampling cost is given by the following equation:

$$C_{\text{RS}} = N \cdot r_{\max} \cdot f \cdot (T_{\text{page}} + T_{\text{cpu-io}} + T_{\text{sample}}) \tag{5}$$

Sample transmission (ST). The samples selected from each processor need to be sent to the coordinator. Each value of the spatial attribute needs $D \cdot S_{\text{number}}$ bytes for representation. Therefore, a total of $N \cdot f \cdot D \cdot S_{\text{number}}$ bytes need to be transmitted through the interconnection, and

$$C_{\text{ST}} = T_{\text{trans}}(N \cdot f \cdot D \cdot S_{\text{number}}) \tag{6}$$

Space partitioning (SP). We assume that the available memory at the coordinator is sufficient to hold the sampled data. Therefore, the sampled data are gathered directly in the main memory of the coordinator. Let S be the sample size of all the processors. Initially the sample points are sorted once for each dimension. Before each split, the data variance must be calculated, meaning that all points in the region must be scanned. Every time a split is occurred in a region, two new regions are produced. By assuming the existence of an internal heapsort algorithm, sorting n elements costs $n \cdot \log n$ comparisons and data exchange operations in the worst case. Therefore, the total cost of this phase is

$$C_{\text{SP}} = (2 \cdot S \cdot \log_2(S)) \cdot (T_{\text{comp}} + T_{\text{swap}}) + \left(\sum_{i=0}^{log_2(P)} \frac{S}{2^i} \right) \cdot T_{\text{comp}} \tag{7}$$

Processor assignment (PA). As stated previously, the assignment can be performed by either exploiting a bipartite weighted matching algorithm or using the simpler greedy alternative. For example, in LEDA [23] an $O(n \cdot (m + n \log n))$ matching algorithm is implemented, where n is the number of vertices and m the number of edges in the graph. The complexity gives the number of numeric computations to calculate the matching. In our case, n equals $2 \cdot P$ and m equals P^2.

The greedy algorithm is based on sorting. In order to determine the number of samples per processor for each region we need to scan the samples in each region. Since there are $f \cdot N$ sample points this requires $f \cdot N \cdot T_{\text{comp}}$ time. Assuming an internal heapsort algorithm sorting the regions requires $P \cdot \log_2 P \cdot (T_{\text{comp}} + T_{\text{swap}})$ time. Therefore the total cost for processor assignment is

$$C_{\text{PA}} = f \cdot N \cdot T_{\text{comp}} + P \cdot \log_2 P \cdot (T_{\text{comp}} + T_{\text{swap}}) \tag{8}$$

Region transmission (RT). All processors need to be notified for the region they are going to handle. Therefore, the coordinator formulates pairs of the form (*processorID*, *rect*) where *processorID* is the processor identification and *rect* is the description of the region, which is composed of $2 \cdot D$ numbers (the lower-left and the upper-right corners). A total of P pairs are sent to each processor. [2] This costs

$$C_{\text{RT}} = P \cdot T_{\text{trans}}(P \cdot (S_{\text{number}} + 2 \cdot D)) \tag{9}$$

Load balancing (LB). During this phase, the processors exchange data in order to achieve load balancing. We assume for simplicity that the data in each processor follow a uniform distribution with respect to the spatial attribute. Therefore, processor P_j preserves N_j/P objects and sends $\frac{P-1}{P} \cdot N_j$ objects to other processors. Summing for all processors we get that the total number of preserved objects and the total number of transmitted objects are given by

$$O_{\text{p}} = \sum_{j=1}^{P} \frac{N_j}{P} = \frac{1}{P} \cdot \sum_{j=1}^{P} N_j = \frac{N}{P}$$

$$O_{\text{t}} = \sum_{j=1}^{P} \frac{P-1}{P} \cdot N_j = \frac{P-1}{P} \cdot \sum_{j=1}^{P} N_j = \frac{P-1}{P} \cdot N$$

The load balancing phase is decomposed to three subphases. First, the processors perform a sequential scan of their data in order to determine the data that must be sent to other processors. The cost of this subphase is dominated by the processor which contains the largest amount of data:

$$C_{\text{read}} = T_{\text{i/o}}\left(\left\lceil \frac{N \cdot r_{\max} \cdot f \cdot S_{\text{rec}}}{S_{\text{cluster}} \cdot S_{\text{page}}} \right\rceil\right)$$

Second, for each object that must be transmitted, processor P_j sends the identification of P_j, the record identifier and the associated spatial attribute. Therefore,

[2] If the system supports broadcasting, this operation can be performed in a single step (i.e. by sending the message(s) to all the processors simultaneously).

$2 \cdot S_{\text{number}} + D \cdot S_{\text{number}}$ bytes are transmitted per record. The cost of data transmission is given by the following equation:

$$C_{\text{trans}} = T_{\text{trans}}\left(\frac{(P-1) \cdot N \cdot (2 \cdot S_{\text{number}} + D \cdot S_{\text{number}})}{P}\right)$$

Finally, each processor writes the data to the disk. This cost is dominated by the processor that contains the largest amount of data after data exchange.

$$C_{\text{write}} = T_{\text{i/o}}\left(\left\lceil \frac{N \cdot r_{\text{skew}}}{P \cdot S_{\text{page}} \cdot S_{\text{cluster}}} \cdot (2 \cdot S_{\text{number}} + D \cdot S_{\text{number}}) \right\rceil\right)$$

Therefore, the total cost of this phase equals:

$$C_{\text{LB}} = C_{\text{read}} + C_{\text{trans}} + C_{\text{write}} \tag{10}$$

Local index construction (LIC). We assume that the cost of this phase is equivalent to external sorting of the data. For example, in [18] the bulk-loading technique proposed is based on external sorting, where the spatial objects are sorted with respect to the Hilbert value of the objects' centroids. The methods proposed in [21,27] are also based on external sorting. External mergesort can be used to fulfill the sorting requirements, as it is described in [14,28]. Therefore, a number of level-0 sorted runs are first generated, and then these runs are merged in order to produce a number of level-1 sorted runs. This procedure is repeated until the whole data collection is sorted in a single run. Evidently, the size of the available main memory is very important, and it should be included in the cost model.

If S_{data} denotes the size of the local data in pages of the dominating processor, then:

$$S_{\text{data}} = \left\lceil \frac{N \cdot r_{\text{skew}}}{P} \cdot \frac{2 \cdot S_{\text{number}} + D \cdot S_{\text{number}}}{S_{\text{page}}} \right\rceil$$

The number of level-0 runs is given by

$$U = \left\lceil \frac{S_{\text{data}}}{S_{\text{mem}}} \right\rceil$$

The number of data points in each run is given by

$$\#\text{pointsPerRun} = S_{\text{mem}} \cdot \left\lfloor \frac{S_{\text{page}}}{S_{\text{number}} \cdot (D+2)} \right\rfloor$$

Assuming an internal heapsort algorithm for sorting the data, the CPU cost to sort the level-0 runs is

$$C_{\text{CPU level-0}} = U \cdot (\#\text{pointsPerRun}) \cdot \log_2(\#\text{pointsPerRun}) \cdot (T_{\text{comp}} + T_{\text{swap}})$$

The number of clusters contained in each level-0 run is given by

$$\#\text{clustersPerRun} = \left\lfloor \frac{S_{\text{data}}}{S_{\text{cluster}} \cdot U} \right\rfloor$$

Therefore, the time to read and write the level-0 runs is

$$C_{\text{i/o level-0}} = U \cdot 2 \cdot T_{\text{i/o}}(\#\text{clustersPerRun})$$

Due to external m-way mergesort, the number of merging steps performed is given by:

$$\#\text{merges} = 1 + \lceil \log_{\#\text{clustersPerRun}-1}(U) \rceil$$

The I/O cost for the merging phase is

$$C_{\text{i/o merge}} = \#\text{merges} \cdot T_{\text{i/o}}\left(2 \cdot \left\lceil \frac{S_{\text{data}}}{S_{\text{cluster}}} \right\rceil\right)$$

Since every point is moved to an output run #merges times and every time #clustersPerRun − 1 comparisons are performed, the CPU cost for the merging phase is

$$C_{\text{CPU merge}} = \frac{r_{\text{skew}} \cdot N}{P} \cdot \#\text{merges} \cdot ((\#\text{clustersPerRun} - 1) \cdot T_{\text{comp}} + T_{\text{move}})$$

By summing the above I/O and CPU costs the final cost of this phase is obtained

$$C_{\text{LIC}} = C_{\text{CPU level-0}} + C_{\text{i/o level-0}} + C_{\text{CPU merge}} + C_{\text{i/o merge}} \tag{11}$$

Global index composition (GIC). The last phase of the algorithm involves the composition of the global index. Each processor reads and transmits the upper levels of the local index to the coordinator. The leaf level of each local index is not transmitted. Finally, the coordinator composes the global index. The cost of the first subphase is dominated by the processor that contains the largest amount of points, because the corresponding local index contains more nodes. The corresponding costs are as follows:

$$C_{\text{read}} = T_{\text{i/o}}\left(\left\lceil \frac{upperNodes_{\max}}{S_{\text{cluster}}} \right\rceil\right)$$

$$C_{\text{trans}} = T_{\text{trans}}(P \cdot upperNodes_{\text{avg}} \cdot S_{\text{page}})$$

$$C_{\text{write}} = T_{\text{i/o}}\left(\left\lceil \frac{P \cdot upperNodes_{\text{avg}}}{S_{\text{cluster}}} \right\rceil\right)$$

In the above equations, $upperNodes_{\text{avg}}$ is the average number of R-tree nodes per processor and $upperNodes_{\max} = nodes_{\text{avg}} \cdot r_{\text{skew}}$ is the maximum number of R-tree nodes that depends on the skew ratio r_{skew}. For brevity, we do not include the formula that gives the number of R-tree nodes. We just mention that it is a function of the tree fanout, the page capacity and the total number of objects (see [12]).

$$C_{\text{GIC}} = C_{\text{read}} + C_{\text{trans}} + C_{\text{write}} \tag{12}$$

If we combine the equations from (8)–(12), the total cost of the parallel bulk-loading algorithm is determined as a function of the various parameters (e.g., number of processors, sampling factor). The total cost is determined by summing-up the costs of the individual phases. This leads to a simple formula that can be adjusted according to the characteristics of the parallel system.

Moreover, "what-if" scenarios can be investigated, by changing the values of the parameters that participate in the cost model. For example, if the values of all parameters except the sampling factor are given, the optimal sampling factor can be determined in order to minimize the overall elapsed time. This can be performed by setting the first derivative of the cost function to zero and determining the value of r_{skew}. The impact of other parameters can be investigated as well. For example, the importance of the network bandwidth, the processor memory size, the database size, the number of processors, the space dimensionality can be studied by varying the respective parameter, keeping the rest constant. Several of these issues are covered in Section 4 which studies the performance of the parallel bulk-loading algorithm.

3.3. Design and implementation alternatives

The first issue that must be addressed is the algorithm for space decomposition. The two fundamental steps are the selection of the split axis and the determination of the split position. The split axis can be selected by using either the *maximum extend* heuristic or the *maximum variance* heuristic. Usually, the *maximum variance* heuristic produces better space decomposition, meaning that the produced regions cover approximately the same volume of the space. In our implementation, we used both heuristics. However, the impact to the overall performance was not significant.

The second important issue is the selection of the bulk-loading algorithm in each processor. The selection of this algorithm has an important impact on the efficiency of the local index construction phase, and also to the quality of the produced global index. Methods that are based on sorting [18,21] are likely to have the same performance with respect to bulk-loading efficiency. The differences with respect to the quality of the global index is anticipated to be equivalent to these for the non-parallel counterparts. However, using bulk-loading methods that avoid sorting, like the ones that are based on buffer trees [1,3], is expected to be more efficient with respect to the index construction time. It is noted that the cost model can be modified in order to reflect the cost of the bulk-loading algorithm executed in each processor. In our performance study we used the algorithm reported in [18].

The third issue we raise is the composition of the global index, when the local indexes are of different height. We propose three solutions to this problem.

(1) The coordinator inserts underflow nodes (possibly with only one descendant) in order to guarantee that all local indexes have the same height. Then, a common root is created (and some other internal nodes if necessary) in order to link the local indexes.
(2) Each processor is forced to build an index with a specific height. This height can be determined by selecting the minimum among the heights of the local indexes. From the number of records assigned to each processor, this value can be determined. When the *threshold height* in each processor is reached, the remaining records are sent to the coordinator, where insertions are performed one-by-one.
(3) Again, each processor builds an index having the same height. The difference with 2 above is that we allow each processor to pack the remaining records in

pages. Then these pages are sent to the coordinator and bulk insertions [19] are used in order to complete the global index composition.

In our performance evaluation study we exploited the third alternative.

Finally, it is important to discuss how we can handle the case when the number of processors is not a power of two. Two solutions can be applied:

(1) In the first solution the space decomposition algorithm does not change. However, if balanced splits are introduced at the end of the decomposition phase some regions will occupy approximately two times more records than others. This has an impact on the local index construction phase as well as on the composition of the global index.
(2) In the second solution, we introduce non-balanced splits during the space decomposition phase. For example, if there are 12 records and three processors the first split generates two regions R_a and R_b such as R_a contains four records, and R_b contains eight records (non-balanced split). Then, region R_b is split into two regions R_{b1} and R_{b2} each containing four records (balanced split). Therefore, each region contains the same number of records.

4. Performance evaluation

In the sequel, a performance evaluation study of the parallel bulk-loading Algorithm is illustrated. The main objective is to investigate the following issues:

- the comparison between the real and the estimated value of the skew ratio r_{skew},
- the effectiveness of load balancing and the cost of the phases of the algorithm,
- the behavior of the algorithm for different parameters such as the sampling factor, the number of processors, the size of the input.
- the quality of the produced index, by inspecting its performance against queries, and
- the performance comparison between the parallel bulk-loading algorithm and its uniprocessor counterpart.

The following real-life and synthetic datasets are used for the experimentation: (a) SEQ1 contains 63,000 locations (Sequoia project), (b) SEQ2 contains 537,118 locations (Sequoia project), (c) LB contains 54,000 road intersections in Long Beach (Tiger/Line data) and (d) GAUSS contains up to 50,000,000 points following the Gaussian distribution. The datasets are depicted in Fig. 7. We refer to [32,33] for a detailed description of the real-life datasets.

4.1. Accuracy of r_{skew} estimation

In this section we investigate the accuracy of Eq. (2) which relates the number of samples per processor (or alternatively the sampling factor f) with the skew

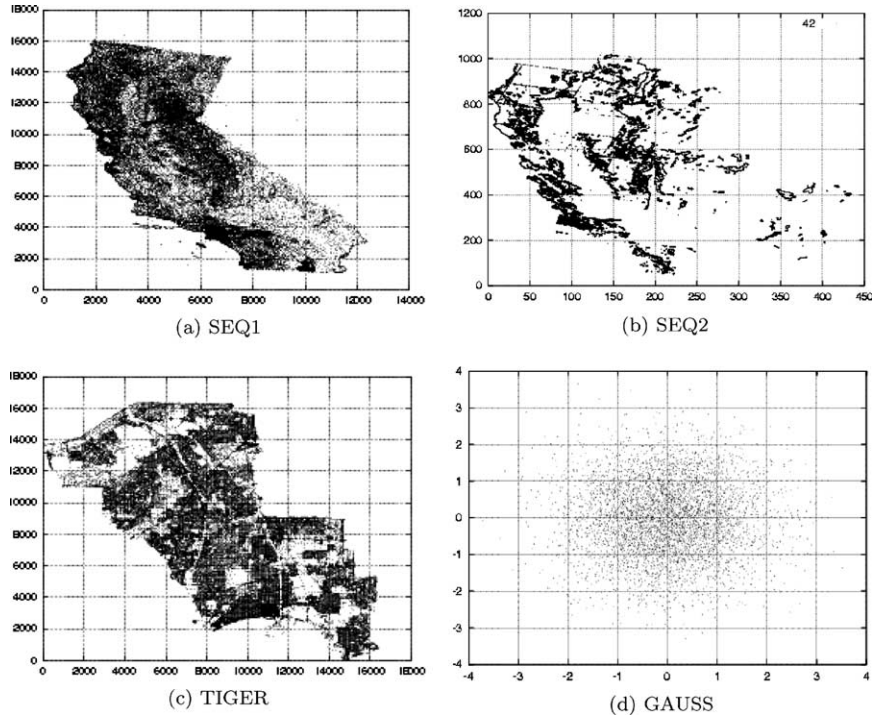


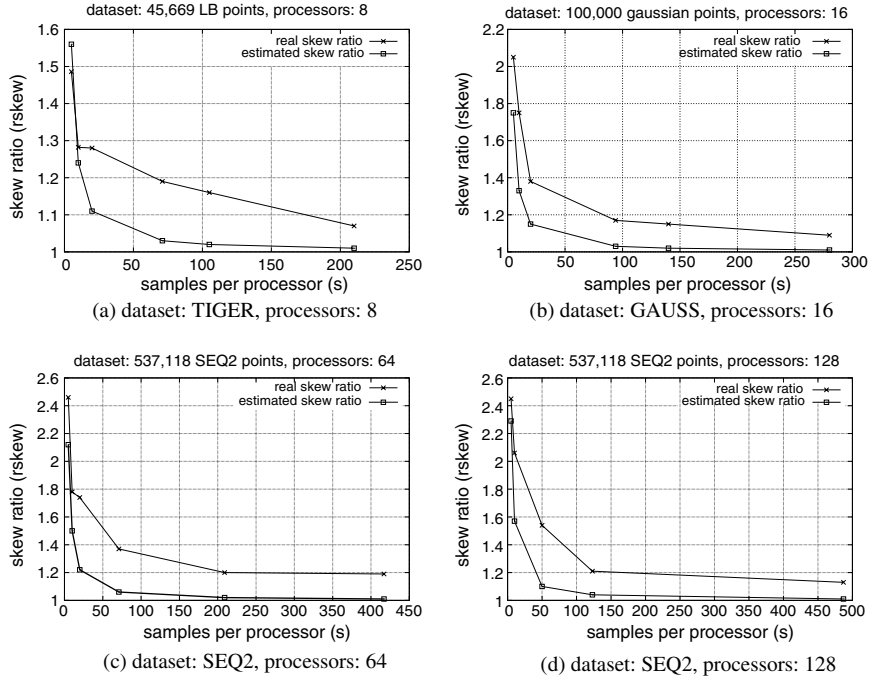
Fig. 7. Datasets used for the experimentation.

ratio r_{skew}. Four separate experiments have been conducted with various datasets and different number of processors. The value for the probability is set to 0.999. The results are illustrated in Fig. 8.

By observing Fig. 8 we obtain that the estimated value is near the real value. The average observed difference is around 17%, which is accurate to a certain degree. The main observation is that the data skew decreases by increasing the number of samples. With more samples per processor, the coordinator performs a better space decomposition with respect to the contents of each region, and therefore the data points are distributed more evenly to the processors. However, as it is shown in the next section, the price paid is that as the number of samples increase the sampling cost becomes significant.

4.2. Experiments

The parallel bulk-loading algorithm and the packed R-tree variant based on the Hilbert space filling curve have been implemented in C/C++. The cost model presented in the previous section is used in order to measure the performance of the proposed algorithm. Therefore, the cost of each operation is determined according to the corresponding formula for the CPU cost, the I/O cost and the network



(a) dataset: TIGER, processors: 8

(b) dataset: GAUSS, processors: 16

(c) dataset: SEQ2, processors: 64

(d) dataset: SEQ2, processors: 128

Fig. 8. Comparison between real and estimated value of r_{skew}.

transmission cost. The real value of r_{skew} is used instead of the analytical one, since the number of objects assigned to each processor is known during the execution of the algorithm.

In order to increase the accuracy of our results, event-driven simulation is used to study the behavior of the proposed algorithm. The query execution cycle is decomposed to several phases that each one contribute to the overall cost of the index construction. An example query execution cycle is illustrated in Fig. 9.

The next issue we study is the impact of the sampling factor to the overall performance of the method. The experiment is conducted for the SEQ2 dataset (Fig. 10(a)) and for a synthetic one (Fig. 10(b)). The synthetic dataset is composed of 2,000,000 points in 2-D space following the Gaussian distribution. The number of processors is set to 8 and the number of buffer pages in each processor is set to 100 (400 KB). By inspecting Fig. 10 the impact of the number of samples to the overall performance is clear. As the number of samples increases, the skew ratio decreases, whereas the sampling cost increases. The overall construction time continues to drop up to a point. After this point, the total cost increases significantly because the costs of random sampling and sample transmission become significant.

The next issue we consider is (a) the speed-up achieved by the parallel bulk loading algorithm (PBULK), and (b) its relative performance with respect to a centralized counterpart (CBULK). The centralized bulk loading algorithm used is the

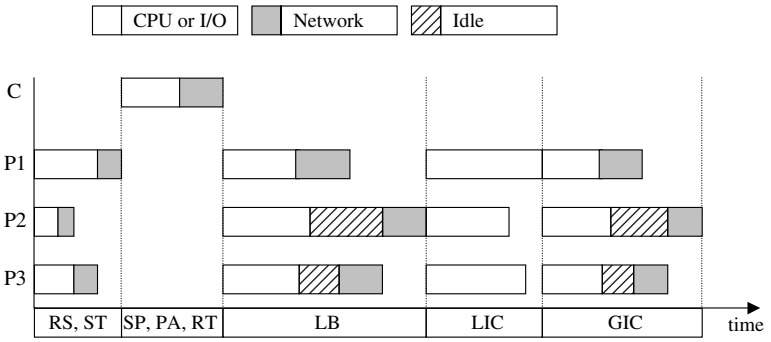
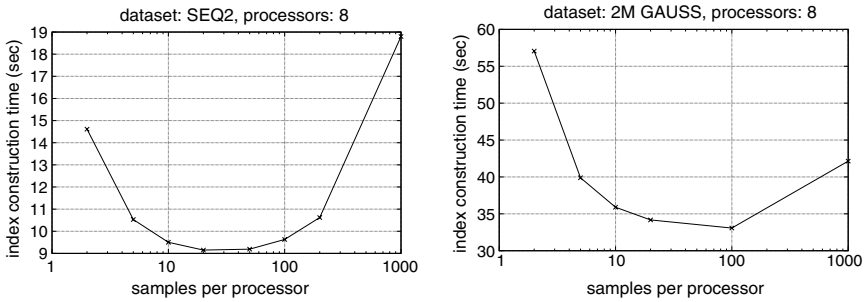


Fig. 9. Query execution cycle example.

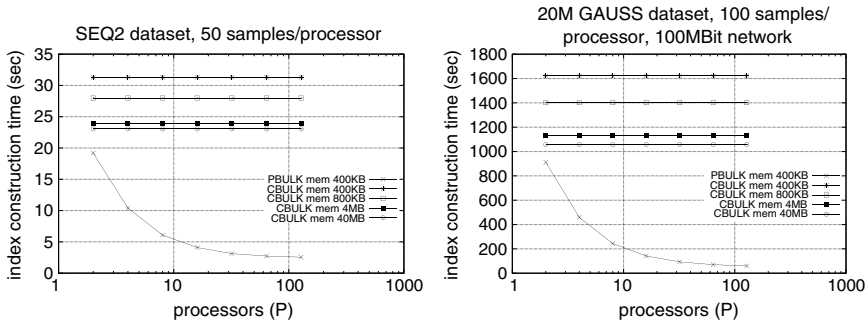


(a) SEQ2 (b) 2M GAUSS

Fig. 10. Index construction time vs. number of samples per processor.

packing method proposed by Kamel and Faloutsos in [18], since this is the method applied by every processor in the parallel algorithm. Alternatively, other bulk-loading methods can be considered, as long as the same method is applied to the parallel algorithm. Fig. 11(a) and (b) illustrate the index construction time with respect to the number of processors, for the SEQ2 dataset and for a Gaussian dataset containing 20,000,000 points. Each processor has 100 pages (400 KB) of available memory, and each processor generates 50 samples. The same graphs illustrate the performance of the centralized algorithm for 100, 200 and 1000 pages of available memory. It is evident that the parallel method outperforms significantly the centralized counterpart. However, if the number of samples increases, then the performance of the centralized method may be marginally better than the parallel one for a small number of processors. This is illustrated in Fig. 12 where each processor produces 500 samples.

Next, we investigate the impact of the database size and the size of the processor memory. It is anticipated that by increasing the database size, more time is required to build the index. The database size affects almost all the phases of the parallel algorithm. The size of the memory is very important, especially for large datasets. The larger the main memory of the processors the less time is required by the local index



(a) SEQ2 (b) GAUSS

Fig. 11. Speed-up for two datasets for 50 samples per processor.

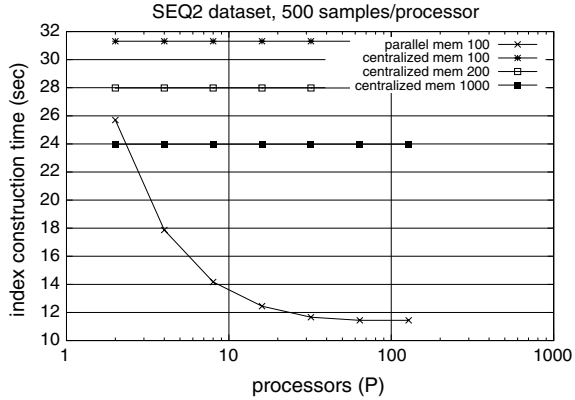
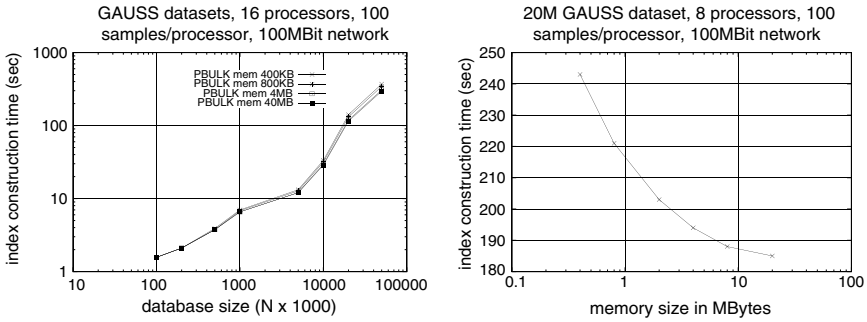


Fig. 12. Speed-up for SEQ2 dataset for 500 samples per processor.

construction phase. The impact of the database size and memory size is depicted in Fig. 13. By increasing the memory size the construction time decreases until we reach the limit in which all local data fit in main memory. Beyond this point, no further improvement is achieved by increasing the memory size.

The next experiment investigates the scale-up capabilities of the method. The tables in Fig. 14 illustrate the total construction time by varying the number of records and the number of processors. The number of samples per processor is 100 and each processor has 4MBytes of available memory. The scale-up ratio ranges between 1.08 and 1.76. By increasing the number of dimensions, more bytes are required to store the value of the spatial attribute, and therefore all phases of the parallel bulk-loading method are affected. However, it is observed that satisfactory results are achieved.

Another important issue that is studied is the impact of the space dimensionality on the performance of the parallel bulk-loading algorithm. It is anticipated that as more dimensions are required to describe the data points, more time is spent on



(a) impact of database size (b) impact of memory size

Fig. 13. Impact of database size and memory size.

Pro cessors	Records	Elapsed Time 2-D (sec)	Elapsed Time 16-D (sec)
2	0.5M	14.53	68.12
4	1M	15.76	74.07
8	2M	18.23	85.96
16	4M	23.17	109.76
32	8M	33.06	157.34
64	16M	52.84	252.51
128	32M	92.39	442.86

Fig. 14. Scale-up for 2-D and 16-D.

the network for transmission, more space is required to store the data and more time is required for each processor to process its local data. In addition, the space partitioning algorithm requires more time to decompose the address space. The impact of the dimensionality is illustrated in Fig. 15.

The last experiment investigates the quality of the produced index with respect to the sampling factor f. Rectangular range queries have been posed to the produced index and the average query execution time has been recorded. Fig. 16 illustrates these results for the TIGER dataset and for 100,000 points in the 5-D space uniformly distributed. The number of processors that have generated the index is 8. The index performance is fairly constant and therefore the index quality does not depend significantly on the sampling factor. However, there is an implicit relationship between the sampling factor and the index quality. If the data skew is large (which is very likely for small sampling factors) then the restructuring procedure presented in Section 3.2 will result in a small increase in the number of R-tree nodes. Consequently, performance degradation takes place whose significance depends on the differences of the tree heights of the local indexes. In our experiments, these differences were not significant, and therefore the index quality remains constant for all tested datasets. A stronger correlation between the sampling factor and the index quality

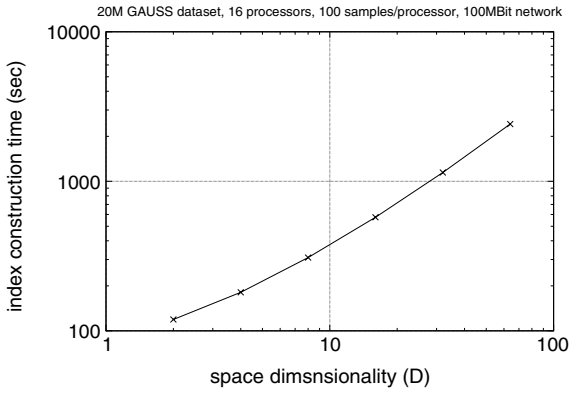
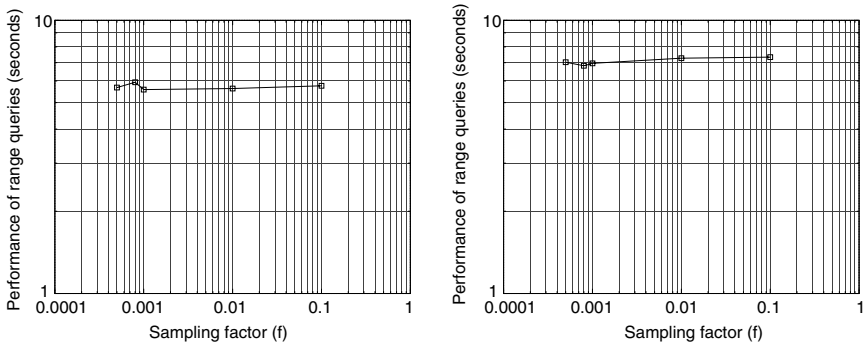


Fig. 15. Impact of database dimensionality.



(a) index quality for TIGER dataset (b) index quality for 5-D uniform dataset

Fig. 16. Index quality vs. sampling factor.

for non-point objects is anticipated, because the overlap in the index nodes is more significant.

5. Concluding remarks and future work

In this work we have studied the problem of parallel bulk-loading multidimensional index structures on a shared-nothing architecture. A methodology has been developed in order to decompose the problem to simpler ones. A cost model has been presented that can be exploited by a query optimizer and aids in predicting the bulk-loading cost. Moreover, experiments have been conducted in order to evaluate the performance of the method for different parameter values (sampling factor, number of processors, dimensionality, etc). The results have shown that the proposed technique is efficient and exploits parallelism to a sufficient degree. The pro-

posed technique can handle cases where the spatial relation is not fragmented to all processors. The load balancing phase guarantees that the *empty* processors will contribute to the local index construction phase. The resulting index can be used in order to answer spatial queries efficiently, or can be exploited in order to redistribute the records according to the values of the spatial attribute. In the latter case, a considerable improvement in terms of query response time is anticipated, as it has been reported in [17,20,25,26]. Further research may include:

- The consideration of non-balanced splits [4] in the space decomposition algorithm. Such a scheme produces regions that do not necessarily contain the same number of objects. Therefore load balancing is sacrificed for the benefit of a better index, especially for large number of dimensions.
- The performance study for non-point objects, where the overlap between index entries is anticipated to be more significant, and therefore the space decomposition algorithm is expected to be crucial to the quality of the produced index.
- The study of overpartitioning techniques where the number of generated space regions is larger than the number of processors. In such a case, more than one region must be assigned to a processor. It is interesting to investigate the impact to the performance of the parallel algorithm.
- The study of the method performance on a network of workstations (NOW), where bulk-loading operations are allowed to be intermixed with other query types.
- Finally, it is interesting to design parallel bulk-loading algorithms for shared-memory architectures, where there is no need to exchange messages among the processors.

References

[1] L. Arge, K.H. Hinrichs, J. Vahrenhold, J.S. Vitter, Efficient bulk operations on dynamic R-trees, Algorithmica 33 (1) (2002) 104–128.
[2] N. Beckmann, H.P. Kriegel, B. Seeger, The R*-tree: an efficient and robust method for points and rectangles, in: Proceedings of 1990 ACM SIGMOD Conference, Atlantic City, NJ, 1990, pp. 322–331.
[3] J. van den Bercken, B. Seeger, P. Widmayer, A generic approach to bulk loading multidimensional index structures, in: Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997, pp. 406–415.
[4] S. Berchtold, C. Bohm, H.-P. Kriegel, Improving the query performance of high-dimensional index structures by bulk load operations, in: Proceedings of the 6th EDBT Conference,, Valencia, Spain, 1998, pp. 216–230.
[5] J. Van den Bercken, B. Seeger, An evaluation of generic bulk loading techniques, in: Proceedings of the 27th VLDB Conference, Roma, Italy, 2001, pp. 461–470.
[6] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, M. Zagha, A comparison of sorting algorithms for the connection machine CM-2, in: Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, 1991.
[7] G. Chartrand, O. Oellermann, Applied and Algorithmic Graph Theory, McGraw-Hill, New York, 1993.
[8] P. Ciaccia, M. Patella, Bulk-loading the M-tree, in: Proceedings of the 9th Australian Database Conference, Perth, Australia, 1998.

[9] D.J. DeWitt et al., The gamma database machine project, IEEE Transactions on Knowledge and Data Engineering 2 (1) (1990) 44–62.
[10] D.J. DeWitt, J.F. Naughton, D.A. Schneider, Parallel sorting on a shared-nothing architecture using probabilistic splitting, in: Proceedings of the 1st PDIS Conference, Miami Beach, FL, 1991, pp. 280–291.
[11] D.J. De Witt, P. Gray, Parallel database systems: the future of high performance database systems, Communications of the ACM 35 (6) (1992) 85–98.
[12] C. Faloutsos, I. Kamel, Beyond uniformity and independence: analysis of R-trees using the concept of fractal dimension, in: Proceedings of the 13th ACM PODS Symposium, Minneapolis, MN, 1994, pp. 4–13.
[13] V. Gaede, O. Guenther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.
[14] G. Graefe, Query evaluation techniques for large databases, ACM Computing Surveys 25 (2) (1993) 73–170.
[15] R.H. Guting, An introduction to spatial database systems, The VLDB Journal 3 (4) (1994) 357–399.
[16] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD Conference, Boston, MA, 1984, pp. 47–57.
[17] I. Kamel, C. Faloutsos, Parallel R-trees, in: Proceedings of 1992 ACM SIGMOD Conference, San Diego, CA, 1992, pp. 195–204.
[18] I. Kamel, C. Faloutsos, On packing R-trees, in: Proceedings of the 2nd CIKM Conference, Washington, DC, 1993, pp. 490–499.
[19] I. Kamel, M. Khalil, V. Kouramajian, Bulk insertion in dynamic R-trees, in: Proceedings of the 4th SDH Symposium, 1996, pp. 3b.31-3b.42.
[20] N. Koudas, C. Faloutsos, I. Kamel, Declustering spatial databases on a multi-computer architecture, in: Proceedings of the EDBT Conference, Avignon, France, 1996, pp. 592–614.
[21] S.T. Leutenegger, M.A. Lopez, J. Edgington, STR: a simple and efficient algorithm for R-tree packing, in: Proceedings of the 13th IEEE ICDE Conference, Birmingham, UK, 1997, pp. 497–506.
[22] S.T. Leutenegger, D.M. Nicol, Efficient bulk-loading of grid files, IEEE Transactions on Knowledge and Data Engineering 9 (3) (1997) 410–420.
[23] K. Melhorn, S. Noher, LEDA: A platform for combinatorial and geometric computing, Communications of the ACM 38 (1) (1995) 96–102.
[24] D. Papadias, Y. Theodoridis, Spatial relations, Minimum bounding rectangles and spatial data structures, Journal of Geographic Information Science 11 (2) (1997) 111–138.
[25] A.N. Papadopoulos, Y. Manolopoulos, Nearest neighbor queries in shared-nothing environments, Geoinformatica 1 (4) (1997) 369–392.
[26] A.N. Papadopoulos, Y. Manolopoulos, Similarity query processing using disk arrays, in: Proceedings of 1998 ACM SIGMOD Conference, Seattle, WA, 1998, pp. 225–236.
[27] N. Roussopoulos, D. Leifker, Direct spatial search on pictorial databases using packed R-trees, in: Proceedings of 1985 ACM SIGMOD Conference, Austin, TX, 1985, pp. 17–31.
[28] B. Salzberg, Merging sorted runs using large main memory, Acta Informatica 27 (3) (1989) 195–215.
[29] T. Sellis, N. Roussopoulos, C. Faloutsos, The R^+-tree: a dynamic index for multidimensional objects, Proceedings of the 13th VLDB Conference, Brighton, UK, 1987, pp. 507–518.
[30] S. Seshadri, J.F. Naughton, Sampling issues in parallel database systems, in: Proceedings of the EDBT '92, 1992, pp. 328–343.
[31] M. Stonebraker, R. Katz, D. Patterson, J. Ousterhout, The design of XPRS, in: Proceedings of the 14th VLDB Conference, Los Angeles, CA, 1988, pp. 318–330.
[32] M. Stonebraker, J. Frew, K. Gardels, J. Meredith, The Sequoia 2000 storage benchmark, in: Proceedings of 1993 ACM SIGMOD Conference, Washington, DC, 1993, pp. 2–11.
[33] TIGER/Line Files, 1994 Technical Documentation, The Bureau of the Census, Washington, DC, 1994.
[34] D.E. Vengroff, J.S. Vitter, I/O Efficient Scientific Computations using TPIE, in: Proceedings of the Goddard Conference on Mass Storage Systems and Technologies, NASA Conference Publication, 1996, pp. 553–570.