High Dimensional Similarity Joins: Algorithms and Performance Evaluation

Nick Koudas, *Member*, *IEEE Computer Society*, and
Kenneth C. Sevcik, *Member*, *IEEE Computer Society*

Abstract—Current data repositories include a variety of data types, including audio, images, and time series. State-of-the-art techniques for indexing such data and doing query processing rely on a transformation of data elements into points in a multidimensional feature space. Indexing and query processing then take place in the feature space. In this paper, we study algorithms for finding relationships among points in multidimensional feature spaces, specifically algorithms for multidimensional joins. Like joins of conventional relations, correlations between multidimensional feature spaces can offer valuable information about the data sets involved. We present several algorithmic paradigms for solving the multidimensional join problem and we discuss their features and limitations. We propose a generalization of the Size Separation Spatial Join algorithm, named *Multidimensional Spatial Join* (MSJ), to solve the multidimensional join problem. We evaluate MSJ along with several other specific algorithms, comparing their performance for various dimensionalities on both real and synthetic multidimensional data sets. Our experimental results indicate that MSJ, which is based on space filling curves, consistently yields good performance across a wide range of dimensionalities.

Index Terms—Spatial join, sort merge joins, multiple-key indexes, data structures.

1 Introduction

Analysis of large bodies of data has become a critical activity in many different contexts. The data types include audio, images, and time series, as well as mixtures of these. A useful and increasingly common way of carrying out this analysis is by using characteristics of data items to associate them with points in a multidimensional feature space so that indexing and query processing can be carried out in the feature space.

Each feature vector consists of d values, which can be interpreted as coordinates in a d-dimensional space, plus some associated content data. Application dependent methods are provided by domain experts to extract feature vectors from data elements and map them into d dimensional space. Moreover, domain experts supply the measure of "similarity" of two entities based on their feature vectors. An important query in this context is the "similarity" query that seeks all points "close" to a specified point in the multidimensional feature space. An additional query of interest is a generalization of the relational join, specifically the multidimensional (or similarity) join query, which reports all pairs of multidimensional points that are "close" (similar) to each other, as measured by some function of their feature value sets.

In a multimedia database that stores collections of images, a multidimensional join query can report pairs of images with similar content, color, texture, etc. The multidimensional join query is useful for both visual exploration and data mining, as well. In a database of stock price information, a multidimensional join query will report all pairs of stocks that are similar to each other with respect to their price movement over a period of time. We evaluate several algorithms for performing joins on high dimensionality data sets.

Agrawal et al. [1] proposed the use of Fourier transforms to map time series data into points in a multidimensional space. The resulting points are inserted into a multidimensional indexing structure. Given a distance predicate ϵ and a query point, all the points within distance ϵ from the query point according to a distance metric can be found by consulting the index. That way, assuming that the query point corresponds to a specific time series and is obtained via the transform, all the time series that are similar to this one will be reported. Faloutsos et al. extend time series similarity to apply to fragments (subsequences) of a time series [10]. Each data sequence is fragmented and sequence fragments are mapped into points in a multidimensional space. That way, subsequence similarity queries can be answered. Agrawal et al. extend previous work on time series similarity to make it robust with respect to the presence of noise and scaling [2]. Queries on multimedia data bases in general are discussed by Faloutsos [8].

This paper is organized as follows: In Section 2, we formalize the problem at hand. In Section 3, we survey and discuss several algorithms for computing multidimensional joins. Section 4 introduces a new algorithm to compute multidimensional joins, named *Multidimensional Spatial Join* (MSJ). Section 5 compares the performance of some of the algorithms experimentally, using both synthetic and real data sets. Finally, Section 6 contains conclusions and a discussion of future work.

- *N. Koudas is with AT&T Research, Florham Park, NJ. E-mail: koudas@research.att.com.*
- *K.C. Sevcik is with the Computer Systems Research Institute, University of Toronto, Canada.*

Manuscript accepted 7 Jan. 1999.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 110163.

2 Problem Statement

We are given two data sets, A and B, containing d-dimensional points of the form $(x_1, x_2, \ldots x_d)$ and $(y_1, y_2, \ldots, y_d)$, respectively. We assume that ranges of all attributes are normalized to the unit interval, [0, 1] so that $0 \leq x_i \leq 1$ and $0 \leq y_i \leq 1$ for $i = 1, \ldots, d$. Given a distance ϵ, a d-dimensional join of A and B contains all pairs of entries $(\mathbf{x}, \mathbf{y}), \mathbf{x} \in A, \mathbf{y} \in B$ such that the distance between them, D_d^ϵ, satisfies

$$D_d^\epsilon = \left(\sum_{i=1}^{d} |(x_i - y_i)^p| \right)^{1/p} \leq \epsilon. \quad (1)$$

Then, D_d^ϵ is referred to as *Manhattan Distance* for $p = 1$ and *Euclidean Distance* for $p = 2$.

Assuming that "good" mapping functions are chosen, objects that are "similar" in the view of the domain experts will map to points that are close in the multidimensional space. The d-dimensional join will report (exactly once) each pair of d-dimensional points, $\mathbf{x} \in A$ and $\mathbf{y} \in B$ that are within distance ϵ from each other according to the chosen distance function. Our goal is to identify efficient algorithms for computing d-dimensional joins for data sets that are much too large to fit in main memory.

In what follows, we adopt a relatively simple model of computation and express all the complexity expressions in terms of the number of operations (comparisons) needed to complete the join query. Alternative models of computation could be adopted, like the one introduced by Aggrawal and Vitter [3], which is based on IO operations.

In the one-dimensional (relational) domain, given two relations, A and B, the join operation $Aj_\theta B$ on attributes a and b applies predicate θ to the attribute values of pairs of tuples and reports a combined tuple whenever the predicate is true. The problem of how to execute joins efficiently is one of the most well-studied problems in data bases. Assuming relations of size n, the worst case complexity of this problem is $O(n^2)$ since all tuples of one relation can possibly match with all the tuples from the joining relation in a degenerate case. However, in some cases, such as where one of the attributes involved in the join is known to be a primary key, the problem can be solved much faster, using, for example, *merge sort*, which is an $O(n \log n)$ algorithm.

The multidimensional join problem as we define it also has a worst case complexity of $O(n^2)$. Consider two d-dimensional point sets A and B, of cardinality n each, such that every point of B is within D_d^ϵ of every point of A for some value of ϵ. In this case, the number of output tuples produced is n^2. Under our definition of the problem, this could happen for any ϵ, provided that both data sets are clustered in the same portion of the multidimensional space. In addition, even when no clustering exists and the points are uniformly distributed in the multidimensional space, for large values of ϵ, the computational work and output size are $O(n^2)$.

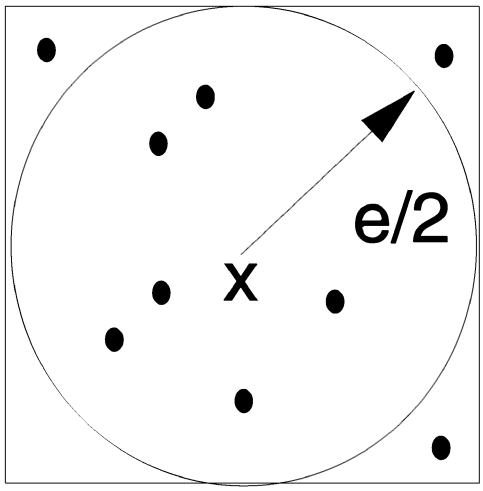

Fig. 1. Approximation of a hyper sphere in two dimensions.

3 Survey of Various Algorithmic Approaches

In this section, we discuss and analyze several algorithms for computing multidimensional joins. We seek algorithms for which efficiency remains high as dimensionality increases. Moreover, since the worst case complexity of the multidimensional join problem is $O(n^2)$, we are interested in identifying instances of the problem that can be solved faster and in comparing the algorithms discussed based on their ability to exploit those instances.

Algorithms for the multidimensional join problem can be separated into two categories. The first category includes algorithms that treat data sets for which indices are not created. The second category includes algorithms that utilize preconstructed indices to solve the multidimensional join problem.

We describe and analyze four algorithmic approaches from the first category that can provide a solution to the multidimensional join problem: *Brute Force*, *Divide and Conquer*, *Replication*, and *Space Filling Curves*. All four algorithmic approaches use the following technique to identify points of interest within distance $\frac{\epsilon}{2}$ of a given point $\mathbf{x}$. Each point $\mathbf{x}$ can be viewed as the center of a hypercube of side ϵ. We refer to the hypercube as the approximation of the multidimensional point. The distance of $\mathbf{x}$ from all the points in the hypercube is computed (based on some distance metric) and only the points within distance ϵ of one another are reported. Let Err denote the error of the approximation of a hypersphere by a hypercube, defined as:

$$Err = Volume(Hypercube) - Volume(Hypersphere). \quad (2)$$

When we approximate a hypersphere of radius $\frac{\epsilon}{2}$ with a hypercube of side ϵ in d dimensions, Err increases with d for constant ϵ and increases with ϵ for constant d. Fig. 1 illustrates the situation in two dimensions. Err corresponds to the area inside the square, but outside the circle. For the rest of this paper, we assume Euclidean distances, but any other distance metric can be applied instead without affecting the operation of the algorithms.

Although the problem of searching and indexing in more than one dimension has been studied extensively, no

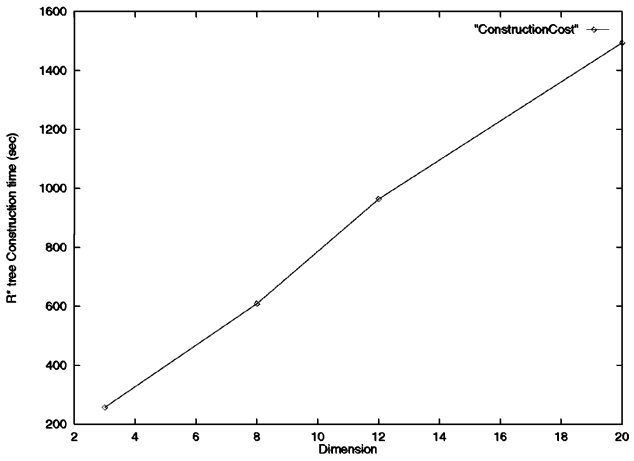

Fig. 2. R* tree construction cost as dimensionality increases.

indexing structure is known that retains its indexing efficiency as dimensionality increases. A wide range of indexing structures have been proposed for the two dimensional indexing problem [22]. Although conceptually, most of these structures generalize to multiple dimensions, in practice, their indexing efficiency degenerates rapidly as dimensionality increases. A recent experimental study by Berchtold et al. [5] showed that, using the X-tree, a multidimensional indexing structure based on R-trees [11], several multidimensional queries degenerate to linear search as dimensionality increases. The X-tree [5] was shown experimentally to outperform previously proposed multidimensional structures, like the TV-tree [13].

An algorithm was proposed by Brinkhoff et al. [6] for the two-dimensional spatial join problem using R^*-trees [7]. Since the R-tree family is a popular family of indexing structures [11], [7], [23], we extended the algorithm of Brinkhoff et al. to multiple dimensions and we report on its performance in subsequent sections. We believe that the trends in performance we report for the join of multidimensional R^*-trees are representative of the join performance of other structures based on the R-tree concepts.

Fig. 2 presents the time to construct an $R^* - tree$ index for 100,000 multidimensional points for various dimensionalities, using multiple insertions. The construction time is measured on a 133MHz processor (including the time to write filled pages into disk). The cost increases approximately linearly as dimensionality increases since the work the algorithm performs per point increases as more dimensions are added. Notice that bulk loading of the index requires application of a multidimensional clustering technique, which has high cost as well. Fig. 2 suggests that, for an on-line solution to the multidimensional join problem, building indices on the fly for nonindexed data sets and using algorithms from the second category to perform the join might not be a viable solution for high dimensionalities due to the prohibitive index construction times.

3.1 Algorithms that Do Not Use Indices

3.1.1 Brute Force Approach

Main Memory Case: If data sets are small enough to fit in main memory together, both can be read into memory and the distance predicate can be evaluated on all pairs of data elements. Assuming A and B are two multidimensional data sets containing n_A and n_B points, respectively, the total cost of this process will be $n_A \times n_B$ predicate evaluations. The cost of each predicate evaluation increases linearly with the dimensionality of the data points. A faster algorithm for the predicate evaluation step is to use a generalization of the *Plane Sweep* technique in multiple dimensions [20]. This makes it possible to reduce the number of distance computations by evaluating the predicate only between pairs of multidimensional points for which the corresponding hypercubes intersect. The complexity of a d-dimensional sweep involving $O(n)$ points, to report k pairs of overlapping objects, is $O(n\ logn^{d-1} + k)$ [16]. Note that if two hypercubes of side $2\delta = \epsilon$ overlap, the points at their centers are not necessarily within distance ϵ of each other. Although the algorithm works well on average, in the worst case, all the pairs of distance computations have to be evaluated at a total cost of $n_A \times n_B$ predicate evaluations plus the overhead of the multidimensional sweep.

Nested Loops (NL): When both data sets cannot fit in main memory, nested loops is the simplest algorithm to apply [25]. Assuming a buffer space of M pages, the total IO cost in page accesses of the join using nested loops will be approximately:

$$|A| + \frac{|A|}{M-1}|B|. \tag{3}$$

Each multidimensional point is approximated with a hypercube and point pairs with intersecting hypercubes are tested for proximity in main memory using a multidimensional sweep. Nested loops can always be applied between two data sets containing $O(n)$ points, but it is an $O(n^2)$ algorithm. The performance of the nested loops algorithm is independent of data distribution, being equally costly for all data distributions. In the relational domain, *Merge Sort* joins and *Hash Joins* have been shown to lead to less costly solutions than nested loops under reasonable statistical assumptions. We investigate analogous alternatives in the multidimensional case.

3.1.2 Divide and Conquer

In this section, we examine two algorithms that are based on the "divide and conquer" algorithmic paradigm. The first one is an application of divide and conquer in multiple dimensions and the second is a recently proposed indexing structure for the multidimensional join problem.

Multidimensional Divide and Conquer Approach (MDC): Multidimensional Divide and Conquer (MDC) is an algorithmic paradigm introduced by Bentley [4] that can be directly applied to the problem at hand. To solve a problem in a multidimensional space, the underlying idea behind the MDC paradigm is to recursively divide the space a dimension at a time and solve the problem in each resulting subspace. Once the problem is solved in all sibling

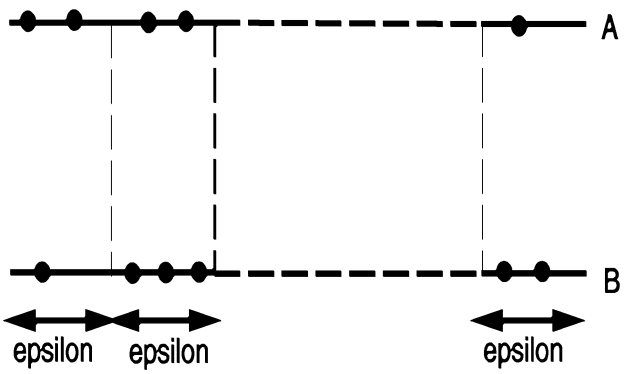

Fig. 3. MDC algorithm for one dimension.

subspaces, then the solutions of the subspaces are combined in a way specific to the problem under consideration.

Consider the one-dimensional case ($d = 1$). Given two sets of n points on a line, we are to report all pairs of points, one from each set, within distance ϵ from each other. We can do this by sorting both sets (an $O(n \log n)$ operation) and performing a scan of both files by treating portions of each file corresponding to a range of values of the attribute of width 2ϵ. As illustrated in Fig. 3, both data sets are sorted on increasing value of the coordinate. By keeping in memory all elements with values in the range 0 to ϵ or ϵ to 2ϵ from both files, we have all the points necessary to correctly report the joining points in the 0 to ϵ range that are part of some joining pair. No more points are necessary since any point that joins with a point in the range 0 to ϵ must be within distance 2ϵ from the left side of the 0 to ϵ range. Once we are done with the 0 to ϵ range, we can discard the corresponding partitions from the buffer pool and read the next range, 2ϵ to 3ϵ, to finish the processing of the ϵ to 2ϵ range, and so on. Corresponding ranges in both files can be processed via the plane sweep algorithm.

Letting $X_{r_1}^{r_2}$ be the number of pages of X that contain at least one value in the range (r_1, r_2), the algorithm is shown in Fig. 4. The cost of sorting n points is $O(n \log n)$ and, assuming that the cost of scanning and joining both files is $O(n)$, the algorithm in Fig. 4 will be an $O(n \log n)$ algorithm. In order for this assumption to be true, the total size of the four partitions $A_{(j-1)\epsilon}^{j\epsilon}, B_{(j-1)\epsilon}^{j\epsilon}, A_{j\epsilon}^{(j+1)\epsilon}, B_{j\epsilon}^{(j+1)\epsilon}$ must be small enough to fit in main memory. More formally:

$$A_{(j-1)\epsilon}^{(j+1)\epsilon} + B_{(j-1)\epsilon}^{(j+1)\epsilon} \leq M \qquad for \quad j = 1 \ldots \left\lceil \frac{1}{\epsilon} \right\rceil. \tag{4}$$

Fig. 5 illustrates the two-dimensional version of the algorithm. Given sets A and B containing points of the form (x, y), the algorithm proceeds as follows: First, sort both files on the y coordinate and partition into two sets A_1, A_2 and B_1, B_2 by splitting on a single value of the y coordinate. Ideally, the partitioning should take place in such a way that $|A_1| = |A_2|$ and $|B_1| = |B_2|$. This can happen if both files have the same median value for each attribute. We apply the algorithm recursively in pairs of corresponding partitions. This means that, for partitions A_1, B_1 (A_2, B_2, respectively), we have to apply the one-dimensional version of the algorithm just described. This involves sorting the partitions on the x coordinate and performing a linear scan, provided that elements in the partitioning corresponding to ranges of size 2ϵ fit in main memory. The area covered by each of these ranges is $2\epsilon^2$. Generalizing this approach to d-dimensional spaces for data sets involving $O(n)$ multidimensional points, will give an $O(n \log^d n)$ [4] algorithm.

Although, in the worst case, all points of a subpartition can be within distance ϵ from a partitioning line, the number is much smaller under reasonable statistical assumptions. Consequently, a variant of this algorithm could store a copy of all the points within distance ϵ from a partitioning line while processing the partitions. Then, we will have to process only these smaller files to correctly

Given two one dimensional data sets, A and B, and ϵ:

- Sort A and B
- Read A_0^ϵ,B_0^ϵ
- Check pairs in A_0^ϵ,B_0^ϵ
- for j = 2 to $\lceil \frac{1}{\epsilon} \rceil$
  1. Read in $A_{(j-1)\epsilon}^{j\epsilon}$,$B_{(j-1)\epsilon}^{j\epsilon}$
  2. Check $A_{(j-2)\epsilon}^{(j-1)\epsilon}$,$B_{(j-1)\epsilon}^{j\epsilon}$
  3. Check $A_{(j-1)\epsilon}^{j\epsilon}$,$B_{(j-2)\epsilon}^{(j-1)\epsilon}$
  4. Discard $A_{(j-2)\epsilon}^{(j-1)\epsilon}$, $B_{(j-2)\epsilon}^{(j-1)\epsilon}$
  5. Check pairs in $A_{(j-1)\epsilon}^{j\epsilon}$,$B_{(j-1)\epsilon}^{j\epsilon}$

Fig. 4. One-dimensional MDC.

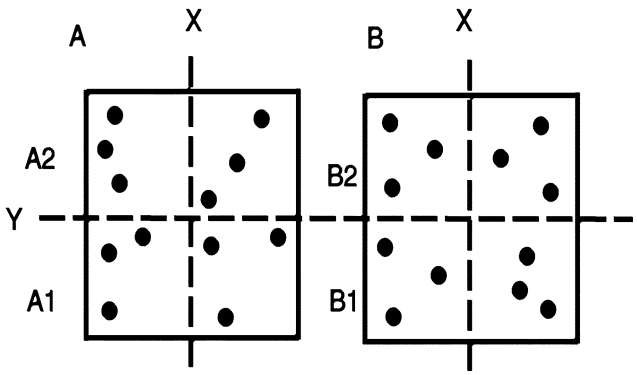

Fig. 5. MDC algorithm for two dimensions.

compute the join, significantly improving the algorithm's performance in most situations.

Although it is conceptually appealing, the application of multidimensional divide and conquer to solve the multidimensional join problem leads to several problems when it is applied in practice. In the general case, the statistical characteristics of the two multidimensional data sets will be different. As a result, partitioning according to the median of a dimension in the first data set might create highly unbalanced partitions for the second. Balanced partitions are necessary in order to attain the complexity of $O(n \log^d n)$ for a problem involving n d-dimensional points. An additional problem is that the constant in the complexity expression is too large: For a d dimensional space, after partitioning according to $d-1$ dimensions, we create 2^{d-1} partitions. Each of these partitions has to be compared against all 2^{d-1} partitions of the joining space. An additional complication is that, in the worst case, the memory space needed for output buffering while partitioning is exponential in the number of dimensions. Multidimensional divide and conquer creates 2^d partitions of the space and, thus, needs 2^d output buffers during the partitioning phase. In summary, we expect that such an approach might be suitable for low dimensionalities and data sets with similar statistical characteristics, but it is not promising as a general solution to the multidimensional join problem.

The ϵ-KDB tree: A new indexing structure for the multidimensional join problem was proposed recently by Shim et al. [24]. The ϵ-KDB tree is intended to speed up the computation of hypercube intersections in main memory.

Given two multidimensional data sets and a distance ϵ, the algorithm proceeds by choosing a dimension and sorting the data sets on this dimension. If both data sets are sorted already on a common dimension, no sorting is necessary. Then, the algorithm proceeds to read the partitions corresponding to intervals of size 2ϵ in dimension of sorted order of both files into main memory and building the ϵ-KDB structure on them. The structure is a variant of KDB trees [21]. It offers a space decomposition scheme that facilitates tree matching since the boundaries of space partitions are canonical. That way, assuming both files have been sorted on a particular dimension, the algorithm can compute the join in time linear in the size of the input data sets by scanning the sorted data sets. In order for the time to be linear, however, the sum of the portions of both A and B in each 2ϵ range along the chosen dimension must fit in main memory. If this is not the case, several problems arise. As new dimensions are introduced on which to perform partitioning, the algorithm must issue a complex schedule of nonsequential page reads from the disk. At each step, the algorithm has to keep neighboring partitions at the same time in main memory. The number of neighboring partitions is exponential in the number of dimensions used for partitioning. Assuming that k dimensions are used for partitioning, in the worst case, we have to keep 2^k partitions from each file being joined in memory at all times. Since the pages holding the partitions are sequentially stored into disk, only two neighboring partitions can be stored sequentially. The rest of the relevant partitions for each step have to be retrieved by scheduling nonsequential IOs.

3.1.3 Replication Approach (REPL)

The replication approach to the multidimensional join problem involves replicating entities and thus causing file sizes to grow. Algorithms based on the replication approach for the two-dimensional problem have been proposed by Patel and DeWitt [19] and Lo and Ravishankar [14]. Here, we explore possible generalizations of these algorithms to higher dimensions. The underlying idea for these algorithms is to divide the two-dimensional space into a number of partitions and then proceed to join corresponding partition pairs. It is possible that the size of a partition pair exceeds the main memory size and, as a result, the pair must be partitioned more finely. During a partitioning phase, any entity that crosses partition boundaries is replicated in all partitions with which it intersects.

For example, in the two-dimensional case, the number of partitions, P, into which to divide the two dimensional space, can be selected as:

$$P = \frac{|A| + |B|}{M}, \tag{5}$$

where $|A|$ and $|B|$ are the sizes of data sets A and B in pages and M is the memory size in pages [19]. Each hypercube that crosses a boundary is replicated in each partition it intersects. This process is repeated for the second data set as well. Once the partitions for both files have been populated and repartitioning has been applied where necessary to make partition pairs fit in main memory, we proceed to join corresponding partition pairs. This strategy correctly computes the join because all the possible joining pairs occur in corresponding partitions. Points located in corresponding partitions form output tuples if they are found to be within ϵ distance of one another.

The algorithm as described generalizes directly to higher dimensions, as shown in Fig. 6. We form a d-dimensional hypercube of side ϵ around each point in both multidimensional spaces and proceed in the same way. Assume that we wish to create 2^{jd} partitions of the space by dividing each of the d dimensions into 2^j intervals, for some j. If points are uniformly distributed in the d-dimensional space, the fraction of hypercubes that fall across partition boundaries is:

$$F = 1 - (1 - \epsilon 2^j)^d, \tag{6}$$

Given two d-dimensional data sets, A and B, and ϵ:

- Select the number of partitions.
- For each data set:
  1. Scan the data set, associating each multidimensional point with the d dimensional hypercube of side ϵ for which the point is the center.
  2. For each hypercube, determine all the partitions to which the hypercube belongs and record the d dimensional point in each such partition.
- Join all pairs of corresponding partitions using multidimensional sweep, repartitioning where necessary.
- Sort the matching pairs and eliminate duplicates

Fig. 6. The REPL Algorithm.

where $\epsilon 2^j$ is the probability that the hypercube is intersected by some plane that partitions the space in one particular dimension. The term $(1-\epsilon 2^j)^d$ expresses the probability that a hypercube is not intersected by any of the 2^{jd} planes that partition the space. Fig. 7 illustrates the amount of replication for various dimensionalities for increasing values of $\epsilon 2^j$.

The problem of replication becomes worse as the dimensionality increases. It is evident that, as dimensionality increases, the fraction of objects replicated increases for a specific value of $\epsilon 2^j$. Intuitively, this can be explained as follows: At dimensionality d, adding a new dimension imposes a partitioning of the $d+1$-dimensional space by 2^j hyperplanes of d dimensions. Consequently, the probability that objects are intersected by the new hyperplanes increases with d. Fig. 8 shows an approximation of the probability that any particular point is replicated (according to the estimate of (6)) as dimensionality increases for various values of ϵ, keeping the number of space partitions constant and equal to 10,000. The probability of replication remains relatively insensitive to the number of dimensions for small values of ϵ. For larger values of ϵ, the probability of replication increases with dimensionality. We have to guarantee some degree of size balance across partitions, otherwise, it might be the case that a large portion of the file falls in one initial partition and extensive repartitioning is required.

There are two major drawbacks to approaches that introduce replication. The appropriate degree of partitioning of the data space is very difficult to choose unless precise statistical knowledge of the multidimensional data sets is provided. Although having such knowledge might

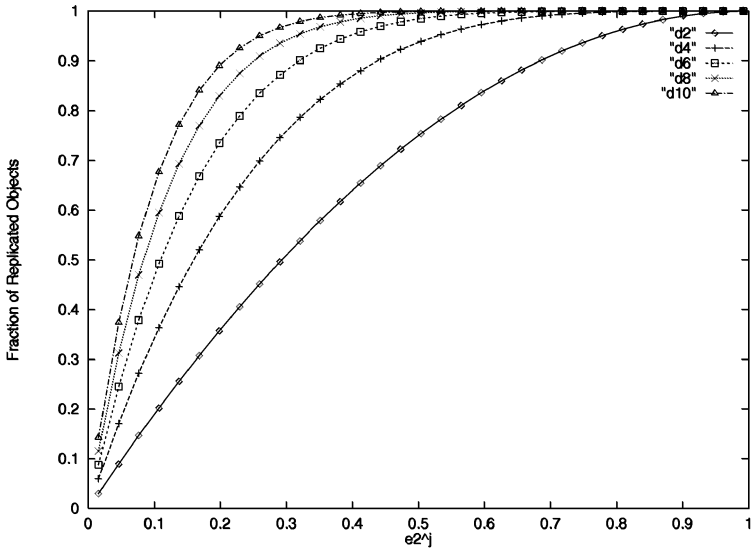

Fig. 7. Fractions of replicated objects for various dimensionalities as a functions of the probability that a hypercube is intersected in one dimension.

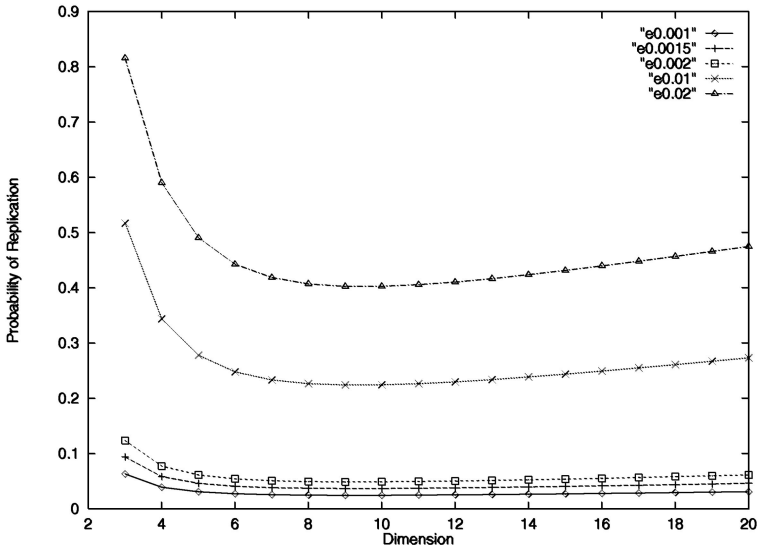

Fig. 8. Probability of Replication using 10,000 partitions for various values of ϵ as a function of dimensionality.

be possible for static multidimensional data sets, it is difficult and costly to obtain for dynamic data sets. Second, when points are relatively dense or ϵ is large, the amount of replication that takes place appears to be very large and it becomes still larger as dimensionality increases.

3.1.4 Space Filling Curves Approach

In this subsection, we explore an algorithm that uses space filling curves to solve the multidimensional join problem.

Orenstein's Algorithm (ZC): Orenstein proposed an algorithm, that we call ZC, to perform joins of multidimensional objects [18]. Starting with multidimensional objects that are approximated by their minimum bounding hypercubes, the hypercubes are tested for intersections. For each pair of intersecting hypercubes, an object intersection test is performed. The algorithm is based on z-curves and their properties. Z-curves reflect a disjoint decomposition of the space into cells and each of the cells is characterized by a bit string. The decimal interpretation of this bit string is called the z-value of the cell. ZC relies on the following property of z-curves to detect intersections: Two approximations of multidimensional objects intersect if and only if their z-values have a prefix-suffix relationship.

The algorithm imposes a recursive binary splitting of the space up to a specific granularity. Each approximated entity is placed in a space partition that fully encloses it. Orenstein [17] presents an analysis of the implication of this decomposition scheme on range query performance and, in subsequent work [18], presents the performance of the multidimensional join algorithm.

This algorithm can be applied to the multidimensional join problem that we address in this paper. In our context, each multidimensional point is approximated with a d-dimensional hypercube of side ϵ. For each multidimensional point, the z value (ZV) at dimensionality d is computed. As dimensionality increases, the processor time to compute the ZV, as well as the number of bits required to store it, increases. We assume that, in a preprocessing step, the ZV associated with each point is computed to some specified precision.

Each data set is scanned and the ZV of each hypercube is transformed to the ZV of the space partition that contains it. The transformation involves setting to zero a number of least significant bits of ZV, depending on the space partition that contains the hypercube. The ZV is a variable length bit string. Shorter bit strings correspond to larger space partitions in the recursive binary decomposition of the space. Both data sets are then sorted into nondecreasing order of ZV values. Then, the algorithm proceeds to merge the two data sets using a stack per data set. At each step, the smaller ZV is selected and processed by comparing it to the ZV at the top of the stack. The algorithm is shown in Fig. 9. The distance predicate is evaluated between pairs of elements whose hypercubes intersect to determine whether the point pair belongs to the final result.

The algorithm as proposed by Orenstein [18] allows for the decomposition of multidimensional objects into a number of pieces. This is useful when the sum of the volumes of hypercubes for each component is significantly less than the volume of the hypercube around the whole object. In our case, however, we deal only with hypercubes around points, so decomposition is not beneficial. Also, decomposing not only increases database size, but also introduces the necessity for a duplicate elimination phase in the algorithm.

3.2 Algorithms that Use Preconstructed Indices

The best known spatial join algorithm for R-trees is the one proposed by Brinkhoff et al. [6]. We have extended it to apply to multidimensional point sets indexed with R^*-trees [7]. The R^*-tree join algorithm is based on an index sweeping process. When the indices have the same height, the algorithm proceeds top-down, sweeping index blocks at the same level. At a specific level, the pairs of overlapping descriptors are identified and, at the same time, the hyperrectangles of their intersections are also computed. This information is used to guide the search in the lower levels since descriptors not overlapping the hyperrectangle

```
For each data set:
        Scan the data set and transform the ZV of the center
        of the hypercube in the space partition that encloses it.
        Sort the data set into nondecreasing order of ZV's.
Initialize to empty a stack per data set ( S_A and S_B )
Loop until either A or B is empty
    Let x_A be the next ZV from A and x_B from B
    if (x_A ≤ x_B) IT ← A; OTHER ← B
    else IT ← B; OTHER ← A
    if x_IT is a prefix of top(S_OTHER) then push(x_IT, S_OTHER)
    else
        loop until either S_OTHER is empty or x_IT becomes a prefix of S_OTHER
            pop(S_OTHER)
        end loop
        evaluate the distance predicate for each pair of overlapping hypercubes.
end loop
```

Fig. 9. The ZC Algorithm.

of intersection of their parents need not be considered for the join. The algorithm uses a buffer pinning technique that follows a greedy approach, trying to keep relevant blocks in the buffer in order to minimize block rereads. When the indices do not have the same height, the algorithm proceeds as described above up to a certain point and then degenerates into a series of range queries.

The multidimensional R^*-tree join algorithm as described can perform the multidimensional similarity join given a distance ϵ as follows: All MBRs of index pages and data pages, as created by the insertion of the multidimensional points, are extended by $\frac{\epsilon}{2}$ in each dimension. The extension is necessary to assure that we do not miss possible joining pairs. The extended MBRs of index pages, as well as data points, are joined using multidimensional sweep.

3.3 Discussion

We have presented two categories of algorithms that can be used to solve the multidimensional join problem. In this paper, we do not include Divide and Conquer algorithms in our experiments due to their known worst case memory and IO requirements. Although MDC will yield an efficient solution for low dimensionalities, it is inapplicable for higher dimensionalities since, in the worst case, it requires a buffer pool size that is exponential in the dimensionality. Similarly, the ϵ-KDB approach will yield very efficient solutions for certain data distributions, but the algorithm's worst case memory requirement and IO complexity are prohibitive for data sets on which partitioning on more than one dimension has to be imposed.

In the next section, we introduce an algorithm, called Multidimensional Spatial Join (MSJ), for the multidimensional join problem. MSJ can use any number of dimensions to decompose the space without affecting its IO cost.

4 Multidimensional Spatial Join (MSJ)

To perform the join of two multidimensional data sets, A and B, we may also use a generalization of the *Size Separation Spatial Join* algorithm (S^3J) [12]. The S^3J algorithm makes use of space filling curves to order the points in a multidimensional space. We assume that the Hilbert value of each multidimensional point is computed at dimensionality d to dL bits of precision, where L is the maximum number of levels of size separation. We consider the Hilbert value computation a preprocessing step of this algorithm. For two d-dimensional data sets, A and B, and given a distance ϵ, we impose a dynamic hierarchical decomposition of the space into *level files*. We scan each data set and place each multidimensional point $(x_1, x_2, \ldots, x_d)$, in a level file l, determined by

$$l = \min_{1 \le i \le d} ncb\Big(x_i - \frac{\epsilon}{2}, x_i + \frac{\epsilon}{2}\Big), \tag{7}$$

where $ncb(b_1, b_2)$ denotes the number of most significant common bits in bit sequences b_1 and b_2. This corresponds to the placement of the approximated multidimensional point in the smallest subpartition of the multidimensional space that fully encloses it. The Hilbert value, H, of each multidimensional point is transformed to the maximum Hilbert value of the space partition that encloses it at level l. This transformation can be achieved by setting to one the $(L - l)d$ least significant bits of H.

The decomposition of the multidimensional space achieved this way provides a flexible way to perform the

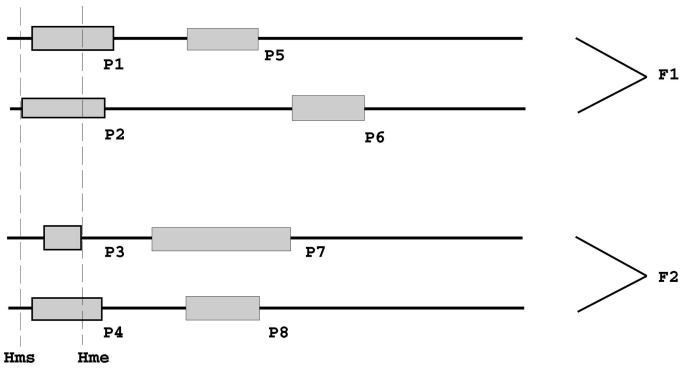

Fig. 10. The multilevel merge phase of MSJ.

multidimensional join [12]. Each subpartition of a level file has to be matched against the corresponding subpartitions in the corresponding and each higher level file of the other data set. That way, in the worst case, we need to keep in memory as many subpartitions for each data set as there are level files. Fig. 11 presents the algorithm. Both data sets are scanned and partitioned into level files. At the same time, Hilbert value transformation takes place. All level files are sorted on the Hilbert value. Finally, a multiway merge of the level files takes place.

Fig. 10 illustrates the merge phase of the algorithm. Two files (F_1 and F_2) have been partitioned into two level files each. At first, the algorithm issues a read of one partition from each level file in main memory, thus partitions P_1, P_2, P_3, P_4 will be read. The minimum starting Hilbert value over all partitions (H_{ms}), as well as the minimum ending value, is computed (H_{me}). Corresponding entries between $[H_{ms}, H_{me}]$ can be processed in main memory. Partitions that are entirely processed (P_3 in Fig. 10) are dropped from the buffer pool and H_{me} is updated. Processing can continue by replacing processed partitions from the corresponding level files (read in P_7 in Fig. 10) and advancing H_{me} as needed, until all level files are processed.

In separating the points in each data set into level files, we may use any subset of the dimensions. The number of dimensions used to separate the input data sets to level files affects the occupancy of each level file. As more dimensions are used, more of the objects occupy higher level files. However, the chance that all objects will be located in one or a few partitions decreases, as more dimensions are used in separating data into level files. Balanced occupancy of space partitions of various levels is desirable. Although, theoretically, an artificial data set can be constructed such that, for a specific value of ϵ, the entire data space falls inside one space partition, the more dimensions we use for partitioning, the less likely this becomes. As is indicated by (7), the computation of the level a point belongs to involves a number of bitwise operations linear in the number of dimensions. All the dimensions can be used for the level computation without significant processor cost.

5 Experimental Evaluation

In this section, we present an experimental evaluation of the performance of MSJ relative to some of the algorithms described in the previous sections for joining of multidimensional data sets.

5.1 Description of Data Sets

For our assessment of the performance of the multidimensional join algorithms, we used both synthetic and real data sets of various dimensionalities. Since the size of each record grows with the number of attributes (dimensions), the overall file size for the fixed number of points increases with the number of dimensions. We choose to keep the number of multidimensional points constant in spaces of different dimensions. An alternative would be to keep the total file size constant by reducing the total number of points as dimensionality increases. However, this would

Given two d-dimensional data sets, A and B, and ϵ distance predicate:

- For each data set:
  1. Scan the data set and partition it into level files, transforming the Hilbert value of the hypercube based on the level file to which it belongs.
  2. Sort the level files into nondecreasing order of Hilbert Values.
- Perform a multi-way merge of all level files

Fig. 11. The MSJ Algorithm.

TABLE 1
Characteristics of Data Sets and Sizes as Dimensionality Increases

Dimension	*D1:100,000 points*	*D2:50,000 points*	*D3:84,640 points*
3/4	3.2 MB	1.6 MB	3.24 MB
8	7 MB	3.5 MB	5.9 MB
12	10 MB	5 MB	8.48 MB
20	16.28 MB	8.14 MB	13.78 MB

create very sparsely populated multidimensional spaces and the performance of multidimensional joins for increasing values of ϵ would be difficult to assess, unless very large file sizes were used.

Table 1 presents the data set sizes in terms of total number of points and total file sizes in bytes at the dimensionalities we experiment with. We keep the buffer pool size constant (2MB) for all experiments.

We perform four series of experiments involving synthetic and real data sets. For each series of experiments, we report two sets of results. In one, we keep ϵ constant and increase the dimensionality of the data set. In the other, we keep the dimensionality of the data set constant and increase the value of ϵ. All of our experiments were conducted on an IBM RS6000 model 43P (133MHz), running AIX with 96MB of main memory with a Seagate Hawk 4 disk with capacity 1GB attached to it. The processor's SPEC ratings are SPECint95 4.72 and SPECfp95 3.76. Average disk access time (including latency) is 18.1 msec assuming random reads. In our implementation, we interleave CPU processing and IO and we bypass the operating system directly using raw IOs. For all the experiments, we performed, we report the response time from the initiation of the operation until its completion. However, explicit times aren't our result. It's relative times of alternative algorithms and approaches that are important and those are insensitive to the details of the underlying system. Instrumenting all the approaches to count IOs and comparisons would be possible, but difficult and unnecessary for our purposes.

The first series of experiments involved multidimensional self-joins between uniformly[1] distributed data sets. The data set used for this experiment has characteristics $D1$. The second experiment involved two data sets, each having characteristics $D2$, containing uniformly distributed multidimensional points generated with different pseudorandom number seeds. Although these experiments involving uniformly distributed data sets offer intuition about the relative performance of the algorithms, it is highly likely that real multidimensional data sets will contain clusters of multidimensional points. These clusters will correspond to groups of entities with similar characteristics. For this reason, in the third series of experiments, we generated multidimensional data sets containing clusters of multidimensional points and we evaluated the performance of the algorithms using the resulting data sets, which again have characteristics D2. The clusters were generated by initializing a kernel in the data space and distributing the points around the cluster kernel using an exponential distribution with mean 0.5. Points outside the unit hypercube were clipped.

Finally, the fourth series of experiments involved actual stock market data on price information collected for 501 companies. We applied a Discrete Fourier Transform (as suggested by Faloutsos [10]) to transform the time series information into points in a multidimensional space. Using a period of 10 days, we extracted several time series from the sequence of prices for each specific stock, obtaining 84,640 multidimensional points. The resulting data set had characteristics $D3$.

5.2 Experimental Results

5.2.1 Experiments with Algorithms Not Based on Preconstructed Indices

Our experiments are summarized in Table 2. The results of these four experiments are presented in Figs. 12, 14, 15, and 16, respectively. Fig. 12 presents the performance of multidimensional self-joins (experiment 1). Note that, for dimensionality $d = 3$, the response time for MSJ is factors of 3, 6, and 20 lower than that of ZC, REPL, and nested loops, respectively. At dimensionality $d = 20$, the corresponding factors are approximately 2, 3, and 30. Fig. 12a presents the performance of the algorithms for $\epsilon = 0.011$. As dimensionality increases, the response time of MSJ increases due to increased sorting cost since the buffer space available to the sort holds smaller and smaller fractions of the data sets. The processor cost increases only slightly with dimensionality since the size of the join result does not change much. At low dimensionality, the size of the join result is a little larger than the size of the input data sets and it decreases to become equal to the size of the input data sets as dimensionality increases. At higher dimensions, a hypersphere of fixed radius inscribes lower percentages of the total space and the probability for a point to match with something more than itself drops rapidly.

Fig. 12b presents the performance of self-joins for increasing values of ϵ at dimensionality $d = 12$. The performance of MSJ appears almost constant since processor time increases only slightly for these values of ϵ, as shown in Fig. 13b. (Processor time increase corresponds to the small increase in the Join phase of MSJ in Fig. 13). IO time for MSJ remains almost constant for the range of ϵ

1. The use of congruential random number generators to create the multidimensional vectors has a distinctive statistical behavior and should be avoided. The resulting multidimensional space has the characteristic that its elements lie mainly on parallel planes [15]. In order to avoid this phenomenon, we use a method to create the multidimensional vectors suggested by Fishman [9].

TABLE 2
Experiments Performed and Characteristics of Data Sets Involved in Each Experiment

Experiment	*Kind of*	*Characteristics*	*% buffer*			
number	*Operation*	*of Data Sets*	*d=3/4*	*d=8*	*d=12*	*d=20*
1	Self Join	D1-uniform	33%	15%	10%	6%
2	Non Self Join	D2-uniform	66%	30%	20%	12%
3	Self Join	D2-clustered	66%	30%	20%	12%
4	Self Join	D3-actual	30.8%	16.9%	11.8%	7.76%

The % buffer column reports the buffer space available to each experiment as a percentage of the total size of the data sets joined for various dimensionalities.

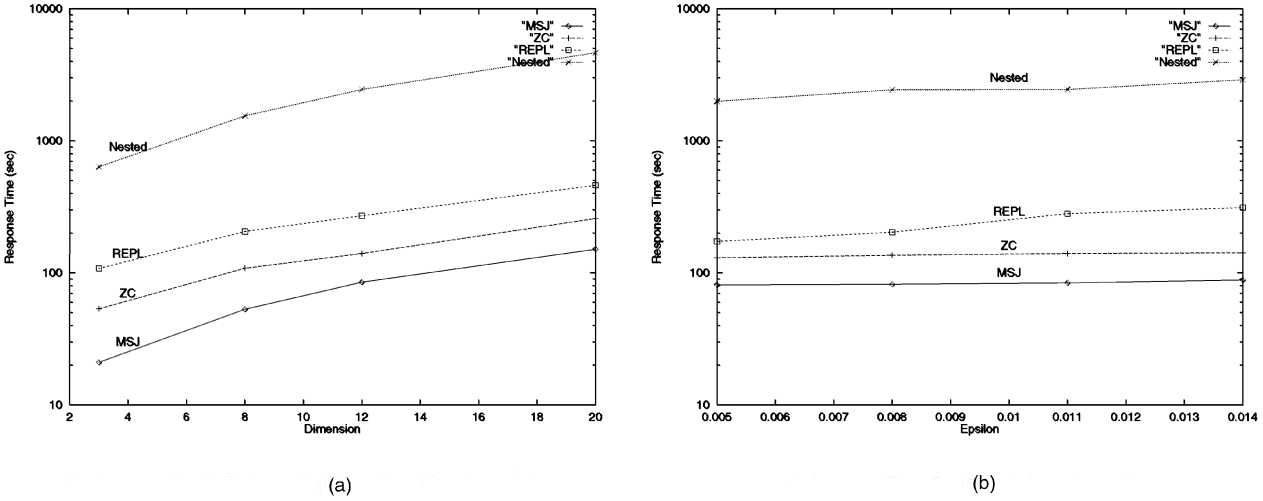

Fig. 12. Performance of self-joins on data sets D1. (a) Increasing dimension for epsilon = 0.011. (b) Increasing epsilon for d = 12.

values of Fig. 12b. For REPL, response time increases due to the increase of replication with ϵ. Similarly, increased replication causes an increase in processor time. For nested loops, processor time increases with ϵ, since the multidimensional sweep in main memory checks more candidate pairs. The join result size for experiment 1 does not change much for the range of ϵ values presented.

Although the IO behavior of MSJ and the ZC algorithm is the same, there are additional processor costs for the ZC algorithm. Fig. 13a presents the portions of time spent in the various phases of the algorithms. The main difference between MSJ and the ZC algorithm is that the sweep process in main memory is data-driven for MSJ, but partition-driven for ZC. ZC relies on the prefix property of the z-curve to perform the join, candidates have to be generated from the stack each time the prefix property of the curve is violated. Violation of the prefix property takes place each time the curve crosses boundaries between different space partitions. Since partitions are seldom full and, thus, are collapsed together in physical pages, this leads to a large amount of data movement in and out from the stacks, as well as plane sweep operations, which constitute an additional processing cost for the algorithm, as is evident from Fig. 13a. Moreover, ZC requires data structure manipulations on the stacks and prefix evaluations for each multidimensional point of the data sets being joined.

For REPL, the amount of replication during the partitioning phase increases with dimensionality (as indicated by (6)) and this increases both processor and IO cost. Processor cost is higher since, by introducing replication, more points are swept. In addition, a duplicate elimination phase has to take place at the end of the algorithm and this involves a sort of the result pairs. Finally, the response time of nested loops increases with dimensionality since relatively less buffer space is available to the operation.

Fig. 14 presents the performance of multidimensional joins between two uniformly distributed data sets of type D2 generated with different seeds (experiment 2). Fig. 14a presents the response times as dimensionality increases for $\epsilon = 0.05$. The join distance predicate is relatively large and, in the case of three dimensions, the join result is large as well. For MSJ, we observe a small decrease in execution time as we move from three to eight dimensions, which can be explained by taking into account the size of the join result. For dimensionality $d = 8$, the size of the join result is much smaller and this explains the difference in execution time. As dimensionality increases, the size of the join result is smaller, but sorting costs increase since the ratio of the buffer pool size to the total input size becomes smaller and

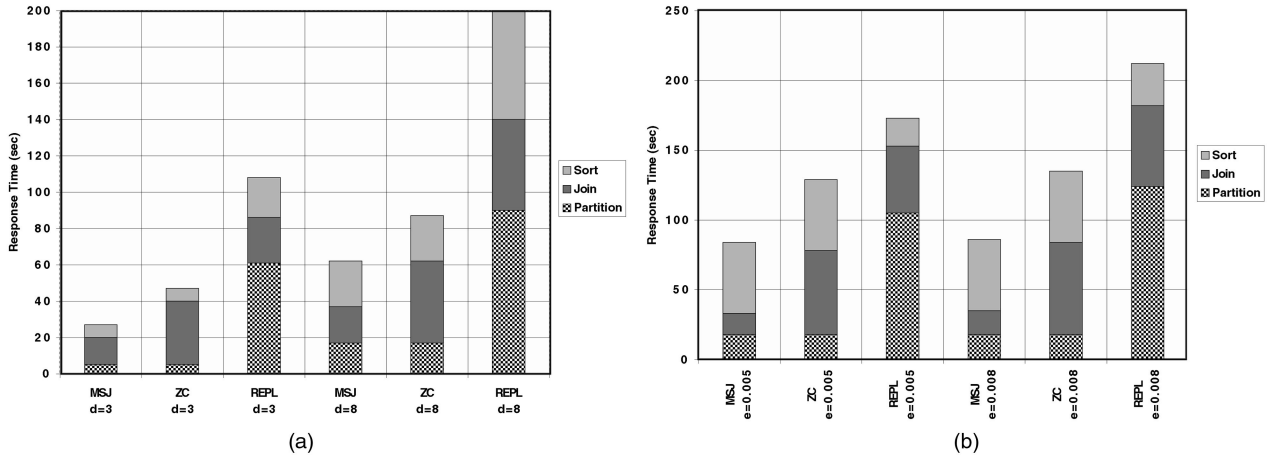

Fig 13. Portion of time spent at different phases of the algorithm. (a) Increasing dimension for epsilon = 0.011. (b) Increasing epsilon for d = 12.

this accounts for the increase in total execution time of MSJ. For the ZC algorithm, we observe a similar behavior. Sorting costs increase as dimensionality increases and, as with the self-join, a larger fraction of processor time is needed to compute the join by ZC relative to MSJ.

For REPL at dimensionality $d = 3$, producing and sorting the join result takes a large fraction of the total execution time. As dimensionality increases, IO time (due to replication and sorting) and processor time (due to sorting and generating additional candidates) increase as well. Replication increases faster than in Fig. 12a because the join predicate is larger (see (6)). For nested loops, IO time, as well as processor time, increases due to the decreasing ratio of buffer space to file size available to the operation and accounts for the increase in execution time as dimensionality increases.

Fig. 14a presents the response time of all algorithms for increasing values of ϵ at dimensionality $d = 12$. For REPL, execution time increases with ϵ due to increased IO time and processor time. The increase is sharper as ϵ increases because larger ϵ values mean that more hypercubes cross boundaries so that more replication occurs. For both algorithms based on space filling curves as well as nested loops, the observations remain the same.

Comparing Fig. 14 to Fig. 12, we see that the performance differences among the algorithms are generally larger for the join of distinct data sets relative to for self-joins. The response time of ZC is about twice that of MSJ quite consistently as ϵ and d are varied. The response time of REPL is about 12 times that of MSJ for all dimensionalities when $\epsilon = 0.05$. It ranges from 4 to 30 times as ϵ goes from 0.01 to 0.10 for dimensionality 12. Finally, Nested loops requires 20 to 40 times times longer than MSJ over the range of dimensionalities for $\epsilon = 0.05$. Fig. 14 shows that, as ϵ increases, the difference in performance between the space filling curve based methods (MSJ and ZC) and the other methods (REPL and Nested loops) increases substantially.

Fig. 15 presents the response time of the algorithms for experiment 3, which involves two data sets containing points that are clustered rather than uniform. The trends in performance for all algorithms for increasing dimensionality are similar to those in the previous experiments (see Fig. 15a). Fig. 15b presents the response time of the algorithms for increasing values of ϵ at dimensionality

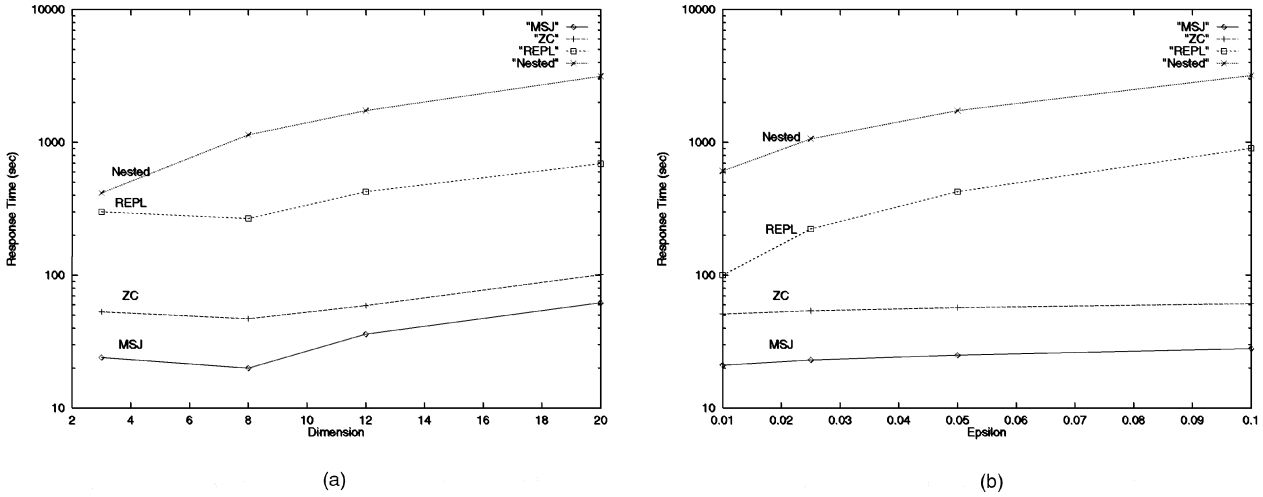

Fig. 14. Performance of multidimensional joins between two distinct uniformly distributed data sets. (a) Increasing dimension for epsilon = 0.05. (b) Increasing epsilon for d = 12.

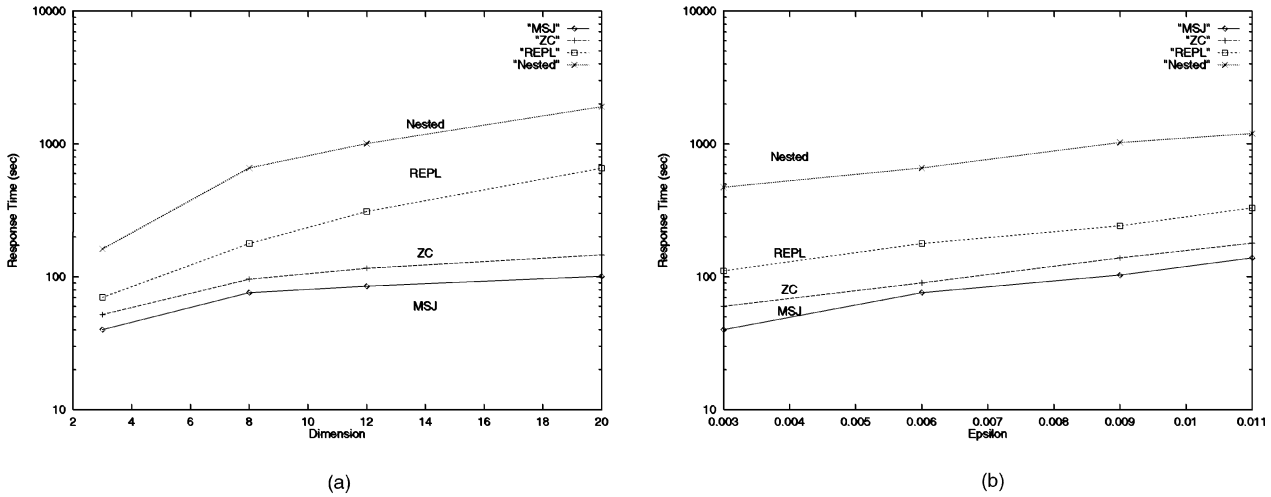

Fig. 15. Performance of multidimensional joins between two distinct clustered data sets. (a) Increasing dimension for epsilon = 0.006. (b) Increasing epsilon for d = 8.

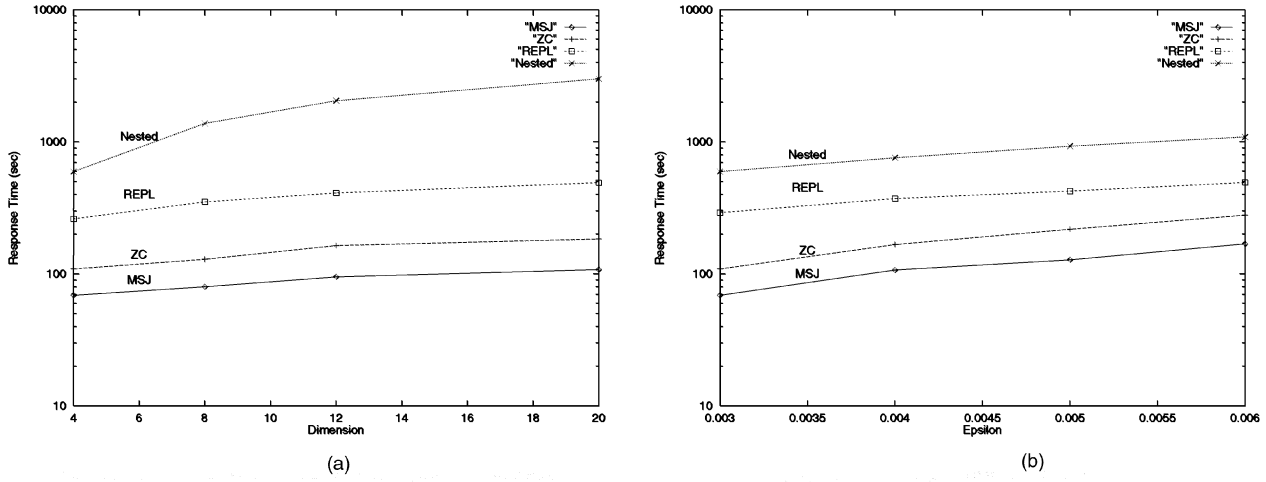

Fig. 16. Performance of joins between stock market data. (a) Increasing dimension for epsilon = 0.003. (b) Increasing epsilon for d=4.

$d = 8$. For all the algorithms, response time increases more rapidly with ϵ than in experiment 1 (for a similar range of ϵ values). Due to clustering, the increase in the size of the join result is larger and, as a result, the processor time needed to compute the candidate and actual pairs increases.

Fig. 16 presents the performance of the algorithms for experiment 4, which involves real stock market data. We employ a multidimensional join operation which reports only the total number of actual joining pairs. (We do not materialize the full join results, due to their size). In Fig. 16a, we present the response time of the algorithms for $\epsilon = 0.03$ as dimensionality increases. For nested loops and REPL, the basic observations are consistent with those from previous experiments. Both algorithms that use space filling curves have increased response times due to their sorting phase as dimensionality increases. However, processor time drops due to the smaller join result size with increasing dimensionality. Both algorithms are processor bound for this experiment and this explains the smoother increase in response time as dimensionality increases.

Fig. 16b presents the response time of the algorithms at dimensionality $d = 4$ for increasing values of ϵ. All algorithms appear to be processor bound and the increase in the join result size accounts for the increase of response times for all algorithms.

Table 3 presents a summary of approximate response time ratios between other algorithms and MSJ as observed in our four experiments. The results are reasonably consistent over the ranges of d and ϵ that we explored. The ZC algorithm had response times between 1.3 and 3 times longer than MSJ over the range of experiments. The REPL algorithm showed more variability in its relative performance, with ratios ranging from 2 to 30 in various cases. Finally, the response times of nested loops were 4 to 100 times larger than MSJ's over the range of cases tested.

5.2.2 Experiments with Algorithms Based on Preconstructed Indices

The experimental results presented for algorithms that don't require preconstructed indices suggest that approaches based on space filling curves and, specifically,

TABLE 3
Summary of Approximate Response Time Ratios of Other Algorithms to MSJ

Ratio	*Exp*	*d varies*		*ϵ varies*	
R_{Nested}/R_{MSJ}	E1	20	30	20	30
	E2	16	40	30	100
	E3	4	20	12	10
	E4	8	30	8	6
R_{REPL}/R_{MSJ}	E1	6	3	2	3
	E2	12	12	4	30
	E3	2	6	3	2.5
	E4	4	3.5	4	2.5
R_{ZC}/R_{MSJ}	E1	3	2	1.5	1.5
	E2	2	2	2.5	2
	E3	1.3	1.3	1.5	1.3
	E4	1.5	1.5	1.5	1.5

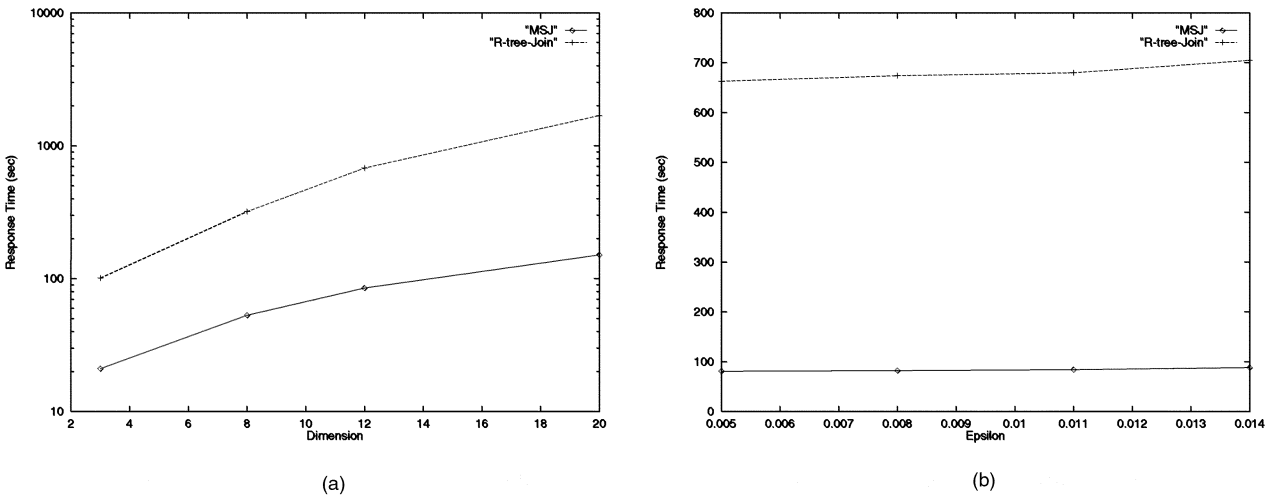

Fig. 17. Performance of multidimensional self-joins: MSJ vs. R^*-tree Join. (a) Increasing dimension for epsilon = 0.011. (b) Increasing epsilon for d = 12.

MSJ, are effective in solving the multidimensional join problem. We also investigate the performance of MSJ in comparison to the algorithms that utilize preconstructed indices.

Since MSJ's approach requires that the Hilbert values of the multidimensional points are precomputed, in this section, we compare the performance of MSJ to that of the R-tree spatial join algorithm (RTJ), assuming that multidimensional R-trees already exist on the data sets involved in the join operation. That is, the cost to construct the multidimensional R-tree indices of the joined data sets is omitted from the performance numbers.

Fig. 17a presents the performance of both MSJ and RTJ for self-join of data sets having characteristics D1 as dimensionality increases. For MSJ, the observations remain exactly the same as those pertaining to Fig. 12. The performance of RTJ deteriorates as dimensionality increases. As dimensionality gets larger, the overlap between R-tree index and leaf entries increases. As a result, the number of pages that have to be pinned in the buffer pool is likely to increase as well. Since the size of the buffer pool is kept constant for varying dimensionalities for both algorithms, the number of page rereads that RTJ has to schedule is expected to increase with dimensionality and this explains the deterioration in performance. The performance of RTJ is very sensitive to the amount of buffering available to the operation. Since the overlap in the R-tree index is expected to increase with dimensionality, the sensitivity gets stronger as dimensionality increases. Fig. 17b presents the performance of both MSJ and RTJ for increasing epsilon and dimensionality $d = 12$. Both algorithms experience an increase in their processor time due to the increasing

number of join tests, for increasing values of epsilon. However, the performance of the RTJ is worse than that of MSJ since it requires a larger number of IOs. We observed similar performance results when comparing MSJ with RTJ for the other data sets used in this study (although we omit the graphs here for brevity).

6 Conclusions

In this paper, we have investigated the problem of computing multidimensional joins between pairs of multidimensional point data sets. We have described several algorithmic approaches that can be applied to the computation of multidimensional joins. There are two main contributions in this work. First, we presented the MSJ algorithm and we experimentally showed that it is a promising solution to the multidimensional join problem. Second, we presented several algorithmic approaches to the multidimensional join problem and discussed their strengths and weaknesses.

Nested loops is applicable in all circumstances, but has a computational complexity that matches the complexity of the multidimensional join problem. Its performance is poor due to redundant processor and IO work, which, for a variety of data distributions, can be avoided. Introducing replication must be done with care, particularly in multiple dimensions. Replication always leads to additional processor and IO work. Our experimental results indicate that algorithms based on space filling curves and, specifically, MSJ, seem promising for computing multidimensional joins across a range of dimensionalities, even if multidimensional R-trees indices already exist for the data sets involved.

Several directions for future work on multidimensional joins are possible. The join result size of a multidimensional join operation is very sensitive to data distributions and to the value of ϵ. For some data distributions, even very small values of ϵ can yield very large result sizes. We feel that multidimensional join queries will be useful in practice only if they can be done interactively. A user will issue a query supplying an ϵ value and, after examining the results, might refine the choice of ϵ. To facilitate this type of interaction, it would be beneficial to restrict the join result size, thus saving a substantial amount of computation for generating the complete list of actual joining pairs. One possible, and useful, restriction would be to report, for each multidimensional point, its k nearest neighbors located at most distance ϵ from the point. Adapting the multidimensional join query to perform this type of computation would be useful.

Both space filling curve approaches discussed in this paper impose a canonical decomposition of the space. Investigation of the effect of alternative multidimensional space organizations in the spirit of the Vantage Point Tree [26] might be helpful in many instances of the problem. Moreover, investigation of randomized algorithms might prove beneficial when approximate results are acceptable. Computational geometry offers many efficient algorithms for the solution of geometric problems in main memory. Investigations of possible applications and generalizations of such algorithms to apply to secondary storage is likely to be a fruitful area of research.

Acknowledgments

This work was performed while Nick Koudas was with the Department of Computer Science, University of Toronto, Canada.

References

[1] R. Agrawal, C. Faloutsos, and A. Swami, "Efficient Similarity Search in Sequence Databases," *Proc. Fourth Int'l Conf. Foundations of Data Organization and Algorithms,* pp. 69-84, Oct. 1993.

[2] R. Agrawal, K. Lin, H.S. Sawhney, and K. Shim, "Fast Similarity Search in the Presence of Noise, Scaling and Translation in Time-Series Databases," *Proc. Very Large Data Bases,* pp. 490-501, Sept. 1995.

[3] A. Aggrawal and J.S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Comm. ACM,* vol. 31, no. 9, pp. 1,116-1,127, 1988.

[4] J.L. Bentley, "Multidimensional Divide-and-Conquer," *Comm. ACM,* vol. 23, no. 4, pp. 214-229, Apr. 1980.

[5] S. Berchtold, D.A. Keim, and H.-P. Kriegel, "The X-Tree: An Index Structure for High Dimensional Data," *Proc. Very Large Data Bases,* pp. 28-30, Sept. 1996.

[6] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins using R-Trees," *Proc. ACM SIGMOD,* pp. 237-246, May 1993.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD,* pp. 220-231, June 1990.

[8] C. Faloutsos, *Indexing Multimedia Databases.* Kluwer, Sept. 1996.

[9] G.S. Fishman, *Concepts and Methods in Discrete Event Digital Simulation.* John Wiley & Sons, 1973.

[10] C. Faloutsos, M. Ranganathan, and I. Manolopoulos, "Fast Subsequence Matching in Time Series Databases," *Proc. ACM SIGMOD,* pp. 419-429, May 1994.

[11] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD,* pp. 47-57, June 1984.

[12] N. Koudas and K.C. Sevcik, "Size Separation Spatial Join," *Proc. ACM SIGMOD,* pp. 324-335, May 1997.

[13] D. Lin, H.V. Jagadish, and C. Faloutsos, "The TV-Tree: A, Index Structure for High-Dimensional Data," *VLDB J.,* vol. 3, no. 4, pp. 517-542, Sept. 1994.

[14] M.-L. Lo and C.V. Ravishankar, "Spatial Hash-Joins," *Proc. ACM SIGMOD,* pp. 247-258, June 1996.

[15] G. Marsaglia, "Random Numbers Fall Mainly in the Planes," *Proc. Nat'l Academy of Science,* vol. 61, pp. 25-28, Sept. 1968.

[16] K. Melhorn, *Data Structures and Algorithms: III, Multidimensional Searching and Computational Geometry.* publisher? June 1991.

[17] J. Orenstein, "Redundancy in Spatial Database," *Proc. ACM SIGMOD,* pp. 294-305, June 1989.

[18] J. Orenstein, "An Algorithm for Computing the Overlay of k-Dimensional Spaces," *Proc. Symp. Large Spatial Databases,* pp. 381-400, Aug. 1991.

[19] J.M. Patel and D.J. DeWitt, "Partition Based Spatial-Merge Join," *Proc. ACM SIGMOD,* pp. 259-270, June 1996.

[20] F.P. Preparata and M.I. Shamos, *Computational Geometry.* New York-Heidelberg-Berlin: Springer-Verlag, Oct. 1985.

[21] J.T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. ACM SIGMOD,* pp. 10-18, 1981.

[22] H. Samet, *The Design and Analysis of Spatial Data Structures.* Addison Wesley, June 1990.

[23] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+ -Tree: A Dynamic Index for Multi-Dimensional Data," *Proc. VLDB 1987,* pp. 507-518, Sept. 1987.

[24] K. Shim, R. Srikant, and R. Agrawal, "High-Dimensional Similarity Joins," *Proc. Int'l Conf. Data Eng.,* also available as IBM Research Report, Apr. 1997.

[25] J.D. Ullman, *Database and Knowledge-Based Systems.* Rockville Md.: Computer Science Press, June 1989.

[26] P. Yianilos, "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces," *Proc. Third Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 311-321, Oct. 1992.

Nick Koudas received his PhD from the University of Toronto in 1998, an MSc from the University of Maryland at College Park in 1994 and a BSc from the University of Patras Computer Engineering and Informatics Department in 1992. He is a senior member of technical staff at the AT&T Shannon Laboratory in Florham Park, New Jersey. He was a postdoctoral researcher at the University of Toronto before joining AT&T Laboratories. His research interests include indexing for advanced database applications, query optimization, data mining, algorithms, performance evaluation, web and databases, and network management. He is a member of the IEEE Computer Society.

Kenneth C. Sevcik holds degrees from Stanford University (BS, mathematics, 1966) and the University of Chicago (PhD, information science, 1971). He is a professor of computer science with a cross-appointment in electrical and computer engineering at the University of Toronto. He is past chairman of the Department of Computer Science, and past director of the Computer Systems Research Institute. His primary area of research interest is in developing techniques and tools for performance evaluation and applying them in such contexts as distributed systems, database systems, local area networks, and parallel computer architectures. Dr. Sevcik served for six years as a member of Canada's Natural Sciences and Engineering Research Council, the primary funding body for research in Science and Engineering in Canada. He is coauthor of the book *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, and co-developer of MAP, a software package for the analysis of queuing network models of computer systems and computer networks. He is a member of the IEEE Computer Society.