Efficient Algorithms for Mining Outliers from Large Data Sets

Sridhar Ramaswamy*
Epiphany Inc.
Palo Alto, CA 94403
sridhar@epiphany.com

Rajeev Rastogi
Bell Laboratories
Murray Hill, NJ 07974
rastogi@bell-labs.com

Kyuseok Shim*
KAIST† and AITrc‡
Taejon, KOREA
shim@cs.kaist.ac.kr

Abstract

In this paper, we propose a novel formulation for distance-based *outliers* that is based on the distance of a point from its k^{th} nearest neighbor. We rank each point on the basis of its distance to its k^{th} nearest neighbor and declare the top n points in this ranking to be outliers. In addition to developing relatively straightforward solutions to finding such outliers based on the classical nested-loop join and index join algorithms, we develop a highly efficient *partition-based* algorithm for mining outliers. This algorithm first partitions the input data set into disjoint subsets, and then prunes entire partitions as soon as it is determined that they cannot contain outliers. This results in substantial savings in computation. We present the results of an extensive experimental study on real-life and synthetic data sets. The results from a real-life NBA database highlight and reveal several expected and unexpected aspects of the database. The results from a study on synthetic data sets demonstrate that the partition-based algorithm scales well with respect to both data set size and data set dimensionality.

1 Introduction

Knowledge discovery in databases, commonly referred to as data mining, is generating enormous interest in both the research and software arenas. However, much of this recent work has focused on finding "large patterns." By the phrase "large patterns", we mean characteristics of the input data that are exhibited by a (typically user-defined) significant portion of the data. Examples of these large patterns include association rules[AMS$^+$95], classification[RS98] and clustering[ZRL96, NH94, EKX95, GRS98].

In this paper, we focus on the converse problem of finding "small patterns" or *outliers*. An outlier in a set of data is an observation or a point that is considerably dissimilar or inconsistent with the remainder of the data. From the above description of outliers, it may seem that outliers are a nuisance—impeding the inference process—and must be quickly identified and eliminated so that they do not interfere with the data analysis. However, this viewpoint is often too narrow since outliers contain useful information. Mining for outliers has a number of useful applications in telecom and credit card fraud, loan approval, pharmaceutical research, weather prediction, financial applications, marketing and customer segmentation.

For instance, consider the problem of detecting credit card fraud. A major problem that credit card companies face is the illegal use of lost or stolen credit cards. Detecting and preventing such use is critical since credit card companies assume liability for unauthorized expenses on lost or stolen cards. Since the usage pattern for a stolen card is unlikely to be similar to its usage prior to being stolen, the new usage points are probably outliers (in an intuitive sense) with respect to the old usage pattern. Detecting these outliers is clearly an important task.

The problem of detecting outliers has been extensively studied in the statistics community (see [BL94] for a good survey of statistical techniques). Typically, the user has to model the data points using a statistical distribution, and points are determined to be outliers depending on how they appear in relation to the postulated model. The main problem with these approaches is that in a number of situations, the user might simply not have enough knowledge about the underlying data distribution. In order to overcome this problem, Knorr and Ng [KN98] propose the following distance-based definition for outliers that is both simple and intuitive: *A point p in a data set is an outlier with respect to parameters k and d if no more than k points in the data set are at a distance of d or less from p*[1]. The distance function can be any metric distance function[2].

The main benefit of the approach in [KN98] is that it does not require any apriori knowledge of data distributions that the statistical methods do. Additionally, the definition of outliers considered is general enough to model statistical

*The work was done while the author was with Bell Laboratories.
†Korea Advanced Institute of Science and Technology
‡Advanced Information Technology Research Center at KAIST

[1]The precise definition used in [KN98] is slightly different from, but equivalent to, this definition.
[2]The algorithms proposed assume that the distance between two points is the euclidean distance between the points.

outlier tests for normal, poisson and other distributions. The authors go on to propose a number of efficient algorithms for finding distance-based outliers. One algorithm is a block nested-loop algorithm that has running time quadratic in the input size. Another algorithm is based on dividing the space into a uniform grid of cells and then using these cells to compute outliers. This algorithm is linear in the size of the database but exponential in the number of dimensions. (The algorithms are discussed in detail in Section 2.)

The definition of outliers from [KN98] has the advantages of being both intuitive and simple, as well as being computationally feasible for large sets of data points. However, it also has certain shortcomings:

1. It requires the user to specify a distance d which could be difficult to determine (the authors suggest trial and error which could require several iterations).

2. It does not provide a ranking for the outliers—for instance a point with very few neighboring points within a distance d can be regarded in some sense as being a *stronger* outlier than a point with more neighbors within distance d.

3. The cell-based algorithm whose complexity is linear in the size of the database does not scale for higher number of dimensions (e.g., 5) since the number of cells needed grows exponentially with dimension.

In this paper, we focus on presenting a new definition for outliers and developing algorithms for mining outliers that address the above-mentioned drawbacks of the approach from [KN98]. Specifically, our definition of an outlier does not require users to specify the distance parameter d. Instead, it is based on the distance of the k^{th} nearest neighbor of a point. For a k and point p, let $D^k(p)$ denote the distance of the k^{th} nearest neighbor of p. Intuitively, $D^k(p)$ is a measure of how much of an outlier point p is. For example, points with larger values for $D^k(p)$ have more sparse neighborhoods and are thus typically stronger outliers than points belonging to dense clusters which will tend to have lower values of $D^k(p)$. Since, in general, the user is interested in the top n outliers, we define outliers as follows: *Given a k and n, a point p is an outlier if no more than $n-1$ other points in the data set have a higher value for D^k than p.* In other words, the top n points with the maximum D^k values are considered outliers. We refer to these outliers as the D^k_n (pronounced "dee-kay-en") outliers of a dataset.

The above definition has intuitive appeal since in essence, it ranks each point based on its distance from its k^{th} nearest neighbor. With our new definition, the user is no longer required to specify the distance d to define the neighborhood of a point. Instead, he/she has to specify the number of outliers n that he/she is in interested in—our definition basically uses the distance of the k^{th} neighbor of the n^{th} outlier to define the neighborhood distance d. Usually, n can be expected to be very small and is relatively independent of the underlying data set, thus making it easier for the user to specify compared to d.

The contributions of this paper are as follows:

- We propose a novel definition for distance-based outliers that has great intuitive appeal. This definition is based on the distance of a point from its k^{th} nearest neighbor.

- The main contribution of this paper is a *partition-based* outlier detection algorithm that first partitions the input points using a clustering algorithm, and computes lower and upper bounds on D^k for points in each partition. It then uses this information to identify the partitions that cannot possibly contain the top n outliers and prunes them. Outliers are then computed from the remaining points (belonging to unpruned partitions) in a final phase. Since n is typically small, our algorithm prunes a significant number of points, and thus results in substantial savings in the amount of computation.

- We present the results of a detailed experimental study of these algorithms on real-life and synthetic data sets. The results from a real-life NBA database highlight and reveal several expected and unexpected aspects of the database. The results from a study on synthetic data sets demonstrate that the partition-based algorithm scales well with respect to both data set size and data set dimensionality. It also performs more than an order of magnitude better than the nested-loop and index-based algorithms.

The rest of this paper is organized as follows. Section 2 discusses related research in the area of finding outliers. Section 3 presents the problem definition and the notation that is used in the rest of the paper. Section 4 presents the nested loop and index-based algorithms for outlier detection. Section 5 discusses our partition-based algorithm for outlier detection. Section 6 contains the results from our experimental analysis of the algorithms. We analyzed the performance of the algorithms on real-life and synthetic databases. Section 7 concludes the paper. The work reported in this paper has been done in the context of the Serendip data mining project at Bell Laboratories (`www.bell-labs.com/projects/serendip`).

2 Related Work

Clustering algorithms like CLARANS [NH94], DBSCAN [EKX95], BIRCH [ZRL96] and CURE [GRS98] consider outliers, but only to the point of ensuring that they do not interfere with the clustering process. Further, the definition of outliers used is in a sense subjective and related to the clusters that are detected by these algorithms. This is in contrast to our definition of distance-based outliers which is more objective and independent of how clusters in the input data set are identified. In [AAR96], the authors address the problem of detecting deviations – after seeing a series of

Symbol	Description
k	Number of neighbors of a point that we are interested in
D^k	Distance of point p to its k^{th} nearest neighbor
n	Total number of outliers we are interested in
N	Total number of input points
δ	Dimensionality of the input
M	Amount of memory available
$dist$	Distance between a pair of points
MINDIST	Minimum distance between a point/MBR and MBR
MAXDIST	Maximum distance between a point/MBR and MBR

Table 1: Notation Used in the Paper

similar data, an element disturbing the series is considered an exception. Table analysis methods from the statistics literature are employed in [SAM98] to attack the problem of finding exceptions in OLAP data cubes. A detailed value of the data cube is called an exception if it is found to differ significantly from the anticipated value calculated using a model that takes into account all aggregates (group-bys) in which the value participates.

As mentioned in the introduction, the concept of distance-based outliers was developed and studied by Knorr and Ng in [KN98]. In this paper, for a k and d, the authors define a point to be an outlier if at most k points are within distance d of the point. They present two algorithms for computing outliers. One is a simple nested-loop algorithm with worst-case complexity $O(\delta N^2)$ where δ is the number of dimensions and N is the number of points in the dataset. In order to overcome the quadratic time complexity of the nested-loop algorithm, the authors propose a cell-based approach for computing outliers in which the δ dimensional space is partitioned into cells with sides of length $\frac{d}{2\sqrt{\delta}}$. The time complexity of this cell-based algorithm is $O(c^\delta + N)$ where c is a number that is inversely proportional to d. This complexity is linear is N but exponential in the number of dimensions. As a result, due to the exponential growth in the number of cells as the number of dimensions is increased, the nested loop outperforms the cell-based algorithm for dimensions 4 and higher.

While existing work on outliers focuses only on the identification aspect, the work in [KN99] also attempts to provide *intensional knowledge*, which is basically an explanation of why an identified outlier is exceptional. Recently, in [BKNS00], the notion of *local* outliers is introduced, which like D^k_n outliers, depend on their local neighborhoods. However, unlike D^k_n outliers, local outliers are defined with respect to the densities of the neighborhoods.

3 Problem Definition and Notation

In this section, we first present a precise statement of the problem of mining outliers from point data sets. We then present some definitions that are used in describing our algorithms. Table 1 describes the notation that we use in the remainder of the paper.

3.1 Problem Statement

Recall from the introduction that we use $D^k(p)$ to denote the distance of point p from its k^{th} nearest neighbor. We rank points on the basis of their $D^k(p)$ distance, leading to the following definition for D^k_n outliers:

Definition 3.1: Given an input data set with N points, parameters n and k, a point p is a D^k_n outlier if there are no more than $n-1$ other points p' such that $D^k(p') > D^k(p)$.[3] ∎

In other words, if we rank points according to their $D^k(p)$ distance, the top n points in this ranking are considered to be outliers. We can use any of the L_p metrics like the L_1 ("manhattan") or L_2 ("euclidean") metrics for measuring the distance between a pair of points. Alternately, for certain application domains (e.g., text documents), nonmetric distance functions can also be used, making our definition of outliers very general.

With the above definition for outliers, it is possible to rank outliers based on their $D^k(p)$ distances—outliers with larger $D^k(p)$ distances have fewer points close to them and are thus intuitively stronger outliers. Finally, we note that for a given k and d, if the distance-based definition from [KN98] results in n' outliers, then each of them is a $D^k_{n'}$ outlier according to our definition.

3.2 Distances between Points and MBRs

One of the key technical tools we use in this paper is the approximation of a set of points using their minimum bounding rectangle (MBR). Then, by computing lower and upper bounds on $D^k(p)$ for points in each MBR, we are able to identify and prune entire MBRs that cannot possibly contain D^k_n outliers. The computation of bounds for MBRs requires us to define the *minimum* and *maximum* distance between two MBRs. Outlier detection is also aided by the computation of the *minimum* and *maximum* possible distance between a point and an MBR, which we define below.

In this paper, we use the square of the euclidean distance (instead of the euclidean distance itself) as the distance metric since it involves fewer and less expensive computations. We denote the distance between two points p and q by $dist(p,q)$. Let us denote a point p in δ-dimensional space by $[p_1, p_2, \ldots, p_\delta]$ and a δ-dimensional rectangle R by the two endpoints of its major diagonal: $r = [r_1, r_2, \ldots, r_\delta]$ and $r' = [r'_1, r'_2, \ldots, r'_\delta]$ such that $r_i \leq r'_i$ for $1 \leq i \leq n$. Let us denote the minimum distance between point p and rectangle R by MINDIST(p, R). Every point in R is at a distance of at least MINDIST(p, R) from p. The following definition of MINDIST is from [RKV95]:

Definition 3.2: MINDIST(p, R) = $\sum_{i=1}^{\delta} x_i^2$, where

[3]Note that more than n points may satisfy our definition of D^k_n outliers—in this case, any n of them satisfying our definition are considered D^k_n outliers.

$$x_i = \begin{cases} r_i - p_i & \text{if } p_i < r_i \\ p_i - r'_i & \text{if } r'_i < p_i \\ 0 & \text{otherwise} \end{cases} \blacksquare$$

We denote the maximum distance between point p and rectangle R by MAXDIST(p, R). That is, no point in R is at a distance that exceeds MAXDIST(p, R) from point p. MAXDIST(p, R) is calculated as follows:

Definition 3.3: MAXDIST(p, R) = $\sum_{i=1}^{\delta} x_i^2$, where

$$x_i = \begin{cases} r'_i - p_i & \text{if } p_i < \frac{r_i + r'_i}{2} \\ p_i - r_i & \text{otherwise} \end{cases} \blacksquare$$

We next define the minimum and maximum distance between two MBRs. Let R and S be two MBRs defined by the endpoints of their major diagonal (r, r' and s, s' respectively) as before. We denote the minimum distance between R and S by MINDIST(R, S). Every point in R is at a distance of at least MINDIST(R, S) from any point in S (and vice-versa). Similarly, the maximum distance between R and S, denoted by MAXDIST(R, S) is defined. The distances can be calculated using the following two formulae:

Definition 3.4: MINDIST(R, S) = $\sum_{i=1}^{\delta} x_i^2$, where

$$x_i = \begin{cases} r_i - s'_i & \text{if } s'_i < r_i \\ s_i - r'_i & \text{if } r'_i < s_i \\ 0 & \text{otherwise} \end{cases} \blacksquare$$

Definition 3.5: MAXDIST(R, S) = $\sum_{i=1}^{\delta} x_i^2$, where $x_i = \max\{|s'_i - r_i|, |r'_i - s_i|\}$. $\blacksquare$

4 Nested-Loop and Index-Based Algorithms

In this section, we describe two relatively straightforward solutions to the problem of computing D_n^k outliers.

Block Nested-Loop Join: The nested-loop algorithm for computing outliers simply computes, for each input point p, $D^k(p)$, the distance of its k^{th} nearest neighbor. It then selects the top n points with the maximum D^k values. In order to compute D^k for points, the algorithm scans the database for each point p. For a point p, a list of the k nearest points for p is maintained, and for each point q from the database which is considered, a check is made to see if $dist(p, q)$ is smaller than the distance of the k^{th} nearest neighbor found so far. If the check succeeds, q is included in the list of the k nearest neighbors for p (if the list contains more than k neighbors, then the point that is furthest away from p is deleted from the list). The nested-loop algorithm can be made I/O efficient by computing D^k for a block of points together.

Index-Based Join: Even with the I/O optimization, the nested-loop approach still requires $O(N^2)$ distance computations. This is expensive computationally, especially if the dimensionality of points is high. The number of distance computations can be substantially reduced by using a spatial index like an R^*-tree [BKSS90].

If we have all the points stored in a spatial index like the R^*-tree, the following pruning optimization, which was pointed out in [RKV95], can be applied to reduce the number of distance computations: Suppose that we have computed $D^k(p)$ for p by looking at a subset of the input points. The value that we have is clearly an upper bound for the actual $D^k(p)$ for p. If the minimum distance between p and the MBR of a node in the R*-tree exceeds the $D^k(p)$ value that we have currently, *none* of the points in the subtree rooted under the node will be among the k nearest neighbors of p. This optimization lets us prune entire subtrees containing points irrelevant to the k-nearest neighbor search for p.[4]

In addition, since we are interested in computing only the top n outliers, we can apply the following pruning optimization for discontinuing the computation of $D^k(p)$ for a point p. Assume that during each step of the index-based algorithm, we store the top n outliers computed. Let D_{nmin} be the minimum D^k among these top outliers. If during the computation of $D^k(p)$ for a point p, we find that the value for $D^k(p)$ computed so far has fallen below D_{nmin}, we are guaranteed that point p cannot be an outlier. Therefore, it can be safely discarded. This is because $D^k(p)$ monotonically *decreases* as we examine more points. Therefore, p is guaranteed to not be one of the top n outliers. Note that this optimization can also be applied to the nested-loop algorithm.

Procedure *computeOutliersIndex* for computing D_n^k outliers is shown in Figure 1. It uses Procedure *getKthNeighborDist* in Figure 2 as a subroutine. In computeOutliersIndex, points are first inserted into an R*-tree index (any other spatial index structure can be used instead of the R*-tree) in steps 1 and 2. The R*-tree is used to compute the k^{th} nearest neighbor for each point. In addition, the procedure keeps track of the n points with the maximum value for D^k at any point during its execution in a heap outHeap. The points are stored in the heap in increasing order of D^k, such that the point with the smallest value for D^k is at the top. This D^k value is also stored in the variable minDkDist and passed to the getKthNeighborDist routine. Initially, outHeap is empty and minDkDist is 0.

The for loop spanning steps 5-13 calls getKthNeighborDist for each point in the input, inserting the point into outHeap if the point's D^k value is among the top n values seen

[4]Note that the work in [RKV95] uses a tighter bound called MINMAXDIST in order to prune nodes. This is because they want to find the maximum possible distance for *the* nearest neighbor point of p, not the k nearest neighbors as we are doing. When looking for the nearest neighbor of a point, we can have a tighter bound for the maximum distance to this neighbor.

```
Procedure computeOutliersIndex($k$,$n$)
begin
1.  for each point $p$ in input data set do
2.      insertIntoIndex(Tree, $p$)
3.  outHeap := $\emptyset$
4.  minDkDist := 0
5.  for each point $p$ in input data set do {
6.      getKthNeighborDist(Tree.Root, $p$, $k$, minDkDist)
7.      if ($p$.DkDist $>$ minDkDist) {
8.          outHeap.insert($p$)
9.          if (outHeap.numPoints() $>$ $n$) outHeap.deleteTop()
10.         if (outHeap.numPoints() $=$ $n$)
11.             minDkDist := outHeap.top().DkDist
12.     }
13. }
14. return outHeap
end
```

Figure 1: Index-Based Algorithm for Computing Outliers

so far (p.DkDist stores the D^k value for point p). If the heap's size exceeds n, the point with the lowest D^k value is removed from the heap and minDkDist updated.

Procedure getKthNeighborDist computes $D^k(p)$ for point p by examining nodes in the R*-tree. It does this using a linked list nodeList. Initially, nodeList contains the root of the R*-tree. Elements in nodeList are sorted, in ascending order of their MINDIST from p.[5] During each iteration of the while loop spanning lines 4–23, the first node from nodeList is examined.

If the node is a leaf node, points in the leaf node are processed. In order to aid this processing, the k nearest neighbors of p among the points examined so far are stored in the heap nearHeap. nearHeap stores points in the decreasing order of their distance from p. p.Dkdist stores D^k for p from the points examined. (It is ∞ until k points are examined.) If at any time, a point q is found whose distance to p is less than p.Dkdist, q is inserted into nearHeap (steps 8–9). If nearHeap contains more than k points, the point at the top of nearHeap discarded, and p.Dkdist updated (steps 10–12). If at any time, the value for p.Dkdist falls below minDkDist (recall that p.Dkdist monotonically decreases as we examine more points), point p cannot be an outlier. Therefore, procedure getKthNeighborDist immediately terminates further computation of D^k for p and returns (step 13). This way, getKthNeighborDist avoids unnecessary computation for a point the moment it is determined that it is not an outlier candidate.

On the other hand, if the node at the head of nodeList is an interior node, the node is expanded by appending its children to nodeList. Then nodeList is sorted according to MINDIST (steps 17–18). In the final steps 20–22, nodes whose minimum distance from p exceed p.DkDist, are pruned. Points contained in these nodes obviously cannot qualify to be amongst p's k nearest neighbors and can be safely ignored.

[5] Distances for nodes are actually computed using their MBRs.

```
Procedure getKthNeighborDist(Root, $p$, $k$, minDkDist)
begin
1.  nodeList := { Root }
2.  $p$.Dkdist := $\infty$
3.  nearHeap := $\emptyset$
4.  while nodeList is not empty do {
5.      delete the first element, Node, from nodeList
6.      if (Node is a leaf) {
7.          for each point $q$ in Node do
8.              if ($dist(p, q) < p$.DkDist) {
9.                  nearHeap.insert($q$)
10.                 if (nearHeap.numPoints() $>$ $k$) nearHeap.deleteTop()
11.                 if (nearHeap.numPoints() $=$ $k$)
12.                     $p$.DkDist := $dist$($p$, nearHeap.top())
13.                 if ($p$.Dkdist $\leq$ minDkDist) return
14.             }
15.     }
16.     else {
17.         append Node's children to nodeList
18.         sort nodeList by MINDIST
19.     }
20.     for each Node in nodeList do
21.         if ($p$.DkDist $\leq$ MINDIST($p$,Node))
22.             delete Node from nodeList
23. }
end
```

Figure 2: Computation of Distance for k^{th} Nearest Neighbor

5 Partition-Based Algorithm

The fundamental shortcoming with the algorithms presented in the previous section is that they are computationally expensive. This is because for each point p in the database we initiate the computation of $D^k(p)$, its distance from its k^{th} nearest neighbor. Since we are only interested in the top n outliers, and typically n is very small, the distance computations for most of the remaining points are of little use and can be altogether avoided.

The partition-based algorithm proposed in this section prunes out points whose distances from their k^{th} nearest neighbors are so small that they cannot possibly make it to the top n outliers. Furthermore, by partitioning the data set, it is able to make this determination for a point p without actually computing the precise value of $D^k(p)$. Our experimental results in Section 6 indicate that this pruning strategy can result in substantial performance speedups due to savings in both computation and I/O.

5.1 Overview

The key idea underlying the partition-based algorithm is to first partition the data space, and then prune partitions as soon as it can be determined that they cannot contain outliers. Since n will typically be very small, this additional preprocessing step performed at the granularity of partitions rather than points eliminates a significant number of points as outlier candidates. Consequently, k^{th} nearest neighbor computations need to be performed for very few points, thus speeding up the computation of outliers. Furthermore, since the number of partitions in the preprocessing step is usually much smaller compared to the number of points, and the preprocessing is performed at the granularity of partitions rather than points, the overhead of preprocessing is low.

We briefly describe the steps performed by the partition-based algorithm below, and defer the presentation of details to subsequent sections.

1. **Generate partitions:** In the first step, we use a clustering algorithm to cluster the data and treat each cluster as a separate partition.

2. **Compute bounds on D^k for points in each partition:** For each partition P, we compute lower and upper bounds (stored in P.lower and P.upper, respectively) on D^k for points in the partition. Thus, for every point $p \in P$, $D^k(p) \geq P$.lower and $D^k(p) \leq P$.upper.

3. **Identify candidate partitions containing outliers:** In this step, we identify the *candidate* partitions, that is, the partitions containing points which are candidates for outliers. Suppose we could compute minDkDist, the lower bound on D^k for the n outliers. Then, if P.upper for a partition P is less than minDkDist, none of the points in P can possibly be outliers. Thus, only partitions P for which P.upper $\geq$ minDkDist are candidate partitions.

   minDkDist can be computed from P.lower for the partitions as follows. Consider the partitions in decreasing order of P.lower. Let $P_1, \ldots, P_l$ be the partitions with the maximum values for P.lower such that the number of points in the partitions is at least n. Then, a lower bound on D^k for an outlier is $\min\{P_i.\text{lower} : 1 \leq i \leq l\}$.

4. **Compute outliers from points in candidate partitions:** In the final step, the outliers are computed from among the points in the candidate partitions. For each candidate partition P, let P.neighbors denote the *neighboring* partitions of P, which are all the partitions within distance P.upper from P. Points belonging to neighboring partitions of P are the only points that need to be examined when computing D^k for each point in P. Since the number of points in the candidate partitions and their neighboring partitions could become quite large, we process the points in the candidate partitions in batches, each batch involving a subset of the candidate partitions.

5.2 Generating Partitions

Partitioning the data space into cells and then treating each cell as a partition is impractical for higher dimensional spaces. This approach was found to be ineffective for more than 4 dimensions in [KN98] due to the exponential growth in the number of cells as the number of dimensions increase.

For effective pruning, we would like to partition the data such that points which are close together are assigned to a single partition. Thus, employing a clustering algorithm for partitioning the data points is a good choice. A number of clustering algorithms have been proposed in the literature, most of which have at least quadratic time complexity [JD88]. Since N could be quite large, we are more interested in clustering algorithms that can handle large data sets. Among algorithms with lower complexities is the pre-clustering phase of BIRCH [ZRL96], a state-of-the-art clustering algorithm that can handle large data sets. The pre-clustering phase has time complexity that is linear in the input size and performs a single scan of the database. It stores a compact summarization for each cluster in a CF-tree which is a balanced tree structure similar to an R-tree [Sam89]. For each successive point, it traverses the CF-tree to find the closest cluster, and if the point is within a threshold distance ϵ of the cluster, it is absorbed into it; else, it starts a new cluster. In case the size of the CF-tree exceeds the main memory size M, the threshold ϵ is increased and clusters in the CF-tree that are within (the new increased) ϵ distance of each other are merged.

The main memory size M and the points in the data set are given as inputs to BIRCH's pre-clustering algorithm. BIRCH generates a set of clusters with generally uniform sizes and that fit in M. We treat each cluster as a separate partition – the points in the partition are simply the points that were assigned to its cluster during the pre-clustering phase. Thus, by controlling the memory size M input to BIRCH, we can control the number of partitions generated. We represent each partition by the MBR for its points. Note that the MBRs for partitions may overlap.

We must emphasize that we use clustering here simply as a heuristic for efficiently generating desirable partitions, and not for computing outliers. Most clustering algorithms, including BIRCH, perform outlier detection; however unlike our notion of outliers, their definition of outliers is not mathematically precise and is more a consequence of operational considerations that need to be addressed during the clustering process.

5.3 Computing Bounds for Partitions

For the purpose of identifying the candidate partitions, we need to first compute the bounds P.lower and P.upper, which have the following property: for all points $p \in P$, P.lower $\leq D^k(p) \leq P$.upper. The bounds P.lower/P.upper for a partition P can be determined by finding the l partitions closest to P with respect to MINDIST/MAXDIST such that the number of points in $P_1, \ldots, P_l$ is at least k. Since the partitions fit in main memory, a main memory index can be used to find the l partitions closest to P (for each partition, its MBR is stored in the index).

Procedure *computeLowerUpper* for computing P.lower and P.upper for partition P is shown in Figure 3. Among its input parameters are the root of the index containing all the

```
Procedure computeLowerUpper(Root, $P$, $k$, minDkDist)
begin
1.  nodeList := { Root }
2.  $P$.lower := $P$.upper := $\infty$
3.  lowerHeap := upperHeap := $\emptyset$
4.  while nodeList is not empty do {
5.      delete the first element, Node, from nodeList
6.      if (Node is a leaf) {
7.          for each partition $Q$ in Node{
8.              if (MINDIST($P$,$Q$) $<$ $P$.lower) {
9.                  lowerHeap.insert($Q$)
10.                 while lowerHeap.numPoints() $-$
11.                         lowerHeap.top().numPoints() $\geq k$ do
12.                     lowerHeap.deleteTop()
13.                 if (lowerHeap.numPoints() $\geq k$)
14.                     $P$.lower := MINDIST($P$, lowerHeap.top())
15.             }
16.             if (MAXDIST($P$,$Q$) $<$ $P$.upper){
17.                 upperHeap.insert($Q$)
18.                 while upperHeap.numPoints() $-$
19.                     upperHeap.top().numPoints() $\geq k$ do
20.                     upperHeap.deleteTop()
21.                 if (upperHeap.numPoints() $\geq k$)
22.                     $P$.upper := MAXDIST($P$, upperHeap.top())
23.                 if ($P$.upper $\leq$ minDkDist) return
24.             }
25.         }
26.     }
27.     else {
28.         append Node's children to nodeList
29.         sort nodeList by MINDIST
30.     }
31.     for each Node in nodeList do
32.         if ($P$.upper $\leq$ MAXDIST($P$,Node) and
33.             $P$.lower $\leq$ MINDIST($P$,Node))
34.             delete Node from nodeList
35. }
end
```

Figure 3: Computation of Lower and Upper Bounds for Partitions

partitions and minDkDist, which is a lower bound on D^k for an outlier. The procedure is invoked by the procedure which computes the candidate partitions, *computeCandidatePartitions*, shown in Figure 4 that we will describe in the next subsection. Procedure computeCandidatePartitions keeps track of minDkDist and passes this to computeLowerUpper so that computation of the bounds for a partition P can be optimized. The idea is that if P.upper for partition P becomes less than minDkDist, then it cannot contain outliers. Computation of bounds for it can cease immediately.

computeLowerUpper is similar to procedure getKthNeighborDist described in the previous section (see Figure 2). It stores partitions in two heaps, lowerHeap and upperHeap, in the decreasing order of MINDIST and MAXDIST from P, respectively – thus, partitions with the largest values of MINDIST and MAXDIST appear at the top of the heaps.

5.4 Computing Candidate Partitions

This is the crucial step in our partition-based algorithm in which we identify the candidate partitions that can potentially contain outliers, and prune the remaining partitions. The idea is to use the bounds computed in the previous section to first estimate minDkDist, which is a lower bound on D^k for an outlier. Then a partition P is a candidate only if P.upper $\geq$ minDkDist. The lower bound minDkDist can be computed using the P.lower values for the partitions as follows. Let $P_1 \ldots, P_l$ be the partitions with the maximum values for P.lower and containing at least n points. Then minDkDist $= \min\{P_i.\text{lower} : 1 \leq i \leq l\}$ is a lower bound on D^k for an outlier.

The procedure for computing the candidate partitions from among the set of partitions PSet is illustrated in Figure 4. The partitions are stored in a main memory index and computeLowerUpper is invoked to compute the lower and upper bounds for each partition. However, instead of computing minDkDist after the bounds for all the partitions have been computed, computeCandidatePartitions stores, in the heap partHeap, the partitions with the largest P.lower values and containing at least n points among them. The partitions are stored in increasing order of P.lower in partHeap and minDkDist is thus equal to P.lower for the partition P at the top of partHeap. The benefit of maintaining minDkDist is that it can be passed as a parameter to computeLowerUpper (in Step 6) and the computation of bounds for a partition P can be halted early if P.upper for it falls below minDkDist. If, for a partition P, P.lower is greater than the current value of minDkDist, then it is inserted into partHeap and the value of minDkDist is appropriately adjusted (steps 8–13).

```
Procedure computeCandidatePartitions(PSet, $k$, $n$)
begin
1.  for each partition $P$ in PSet do
2.      insertIntoIndex(Tree, $P$)
3.  partHeap := $\emptyset$
4.  minDkDist := 0
5.  for each partition $P$ in PSet do {
6.      computeLowerUpper(Tree.Root, $P$, $k$, minDkDist)
7.      if ($P$.lower $>$ minDkDist) {
8.          partHeap.insert($P$)
9.          while partHeap.numPoints() $-$
10.                 partHeap.top().numPoints() $\geq n$ do
11.             partHeap.deleteTop()
12.         if (partHeap.numPoints() $\geq n$)
13.             minDkDist := partHeap.top().lower
14.     }
15. }
16. candSet := $\emptyset$
17. for each partition $P$ in PSet do
18.     if ($P$.upper $\geq$ minDkDist) {
19.         candSet := candSet $\cup$ $\{P\}$
20.         $P$.neighbors :=
21.                 $\{Q$: $Q \in$ PSet and MINDIST($P$,$Q$) $\leq$ $P$.upper$\}$
22.     }
23. return candSet
end
```

Figure 4: Computation of Candidate Partitions

In the for loop over steps 17–22, the set of candidate partitions candSet is computed, and for each candidate partition P, partitions Q that can potentially contain the k^{th}

nearest neighbor for a point in P are added to P.neighbors (note that P.neighbors contains P).

5.5 Computing Outliers from Candidate Partitions

In the final step, we compute the top n outliers from the candidate partitions in candSet. If points in all the candidate partitions and their neighbors fit in memory, then we can simply load all the points into a main memory spatial index. The index-based algorithm (see Figure 1) can then be used to compute the n outliers by probing the index to compute D^k values only for points belonging to the candidate partitions. Since both the size of the index as well as the number of candidate points will in general be small compared to the total number of points in the data set, this can be expected to be much faster than executing the index-based algorithm on the entire data set of points.

In the case that all the candidate partitions and their neighbors exceed the size of main memory, then we need to process the candidate partitions in batches. In each batch, a subset of the remaining candidate partitions that along with their neighbors fit in memory, is chosen for processing. Due to space constraints, we refer the reader to [RRS98] for details of the batch processing algorithm.

6 Experimental Results

We empirically compared the performance of our partition-based algorithm with the block nested-loop and index-based algorithms. In our experiments, we found that the partition-based algorithm scales well with both the data set size as well as data set dimensionality. In addition, in a number of cases, it is more than an order of magnitude faster than the block nested-loop and index-based algorithms.

We begin by describing in Section 6.1 our experience with mining a real-life NBA (National Basketball Association) database using our notion of outliers. The results indicate the efficacy of our approach in finding “interesting” and sometimes unexpected facts buried in the data. We then evaluate the performance of the algorithms on a class of synthetic datasets in Section 6.2. The experiments were performed on a Sun Ultra-2/200 workstation with 512 MB of main memory, and running Solaris 2.5. The data sets were stored on a local disk.

6.1 Analyzing NBA Statistics

We analyzed the statistics for the 1998 NBA season with our outlier programs to see if it could discover interesting nuggets in those statistics. We had information about all 471 NBA players who played in the NBA during the 1997-1998 season. In order to restrict our attention to significant players, we removed all players who scored less then 100 points over the course of the entire season. This left us with 335 players. We then wanted to ensure that all the columns were given equal weight. We accomplished this by transforming the value c in a column to $\frac{c-\bar{c}}{\sigma_c}$ where $\bar{c}$ is the average value of the column and σ_c its standard deviation. This transformation normalizes the column to have an average of 0 and a standard deviation of 1.

We then ran our outlier program on the transformed data. We used a value of 10 for k and looked for the top 5 outliers. The results from some of the runs are shown in Figure 5. (findOuts.pl is a perl front end to the outliers program that understands the names of the columns in the NBA database. It simply processes its arguments and calls the outlier program.) In addition to giving the actual value for a column, the output also prints the normalized value used in the outlier calculation. The outliers are ranked based on their D^k values which are listed under the DIST column.

The first experiment in Figure 5 focuses on the three most commonly used *average* statistics in the NBA: average points per game, average assists per game and average rebounds per game. What stands out is the extent to which players having a large value in one dimension tend to dominate in the outlier list. For instance, Dennis Rodman, not known to excel in either assisting or scoring, is nevertheless the top outlier because of his huge (nearly 4.4 sigmas) deviation from the average on rebounds. Furthermore, his DIST value is much higher than that for any of the other outliers, thus making him an extremely strong outlier. Two other players in this outlier list also tend to dominate in one or two columns. An interesting case is that of Shaquille O’ Neal who made it to the outlier list due to his excellent record in both scoring and rebounds, though he is quite average on assists. (Recall that the average of every normalized column is 0.) The first “well-rounded” player to appear in this list is Karl Malone, at position 5. (Michael Jordan is at position 7.) In fact, in the list of the top 25 outliers, there are only two players, Karl Malone and Grant Hill (at positions 5 and 6) that have normalized values of more than 1 in all three columns.

When we look at more defensive statistics, the outliers are once again dominated by players having large normalized values for a single column. When we consider average steals and blocks, the outliers are dominated by shot blockers like Marcus Camby. Hakeem Olajuwon, at position 5, shows up as the first “balanced” player due to his above average record with respect to both steals and blocks.

In conclusion, we were somewhat surprise by the outcome of our experiments on the NBA data. First, we found that very few “balanced” players (that is, players who are above average in every aspect of the game) are labeled as outliers. Instead, the outlier lists are dominated by players who excel by a wide margin in particular aspects of the game (e.g., Dennis Rodman on rebounds).

Another interesting observation we made was that the outliers found tended to be more interesting when we considered fewer attributes (e.g., 2 or 3). This is not entirely surprising since it is a well-known fact that as the number of dimensions increases, points spread out more uniformly in the data space and distances between them are a poor measure of their similarity/dissimilarity.

```
->findOuts.pl -n 5 -k 10 reb assists pts
          NAME   DIST  avgReb  (norm) avgAssts  (norm)   avgPts  (norm)
  Dennis Rodman  7.26  15.000  (4.376)   2.900  (0.670)   4.700 (-0.459)
 Rod Strickland  3.95   5.300  (0.750)  10.500  (4.922)  17.800  (1.740)
Shaquille Oneal  3.61  11.400  (3.030)   2.400  (0.391)  28.300  (3.503)
Jayson Williams  3.33  13.600  (3.852)   1.000 (-0.393)  12.900  (0.918)
    Karl Malone  2.96  10.300  (2.619)   3.900  (1.230)  27.000  (3.285)

->findOuts.pl -n 5 -k 10 steal blocks
           NAME  DIST    avgSteals (norm)    avgBlocks  (norm)
   Marcus Camby  8.44     1.100    (0.838)     3.700    (6.139)
Dikembe Mutombo  5.35     0.400   (-0.550)     3.400    (5.580)
  Shawn Bradley  4.36     0.800    (0.243)     3.300    (5.394)
   Theo Ratliff  3.51     0.600   (-0.153)     3.200    (5.208)
Hakeem Olajuwon  3.47     1.800    (2.225)     2.000    (2.972)
```

Figure 5: Finding Outliers from a 1998 NBA Statistics Database

Finally, while we were conducting our experiments on the NBA database, we realized that specifying actual distances, as is required in [KN98], is fairly difficult in practice. Instead, our notion of outliers, which only requires us to specify the k-value used in calculating k'th neighbor distance, is much simpler to work with. (The results are fairly insensitive to minor changes in k, making the job of specifying it easy.) Note also the ranking for players that we provide in Figure 5 based on distance —this enables us to determine how strong an outlier really is.

6.2 Performance Results on Synthetic Data

We begin this section by briefly describing our implementation of the three algorithms that we used. We then move onto describing the synthetic datasets that we used.

6.2.1 Algorithms Implemented

Block Nested-Loop Algorithm: This algorithm was described in Section 4. In order to optimize the performance of this algorithm, we implemented our own buffer manager and performed reads in large blocks. We allocated as much buffer space as possible to the outer loop.

Index-Based Algorithm: To speed up execution, an R*-tree was used to find the k nearest neighbors for a point, as described in Section 4. The R*-tree code was developed at the University of Maryland.[6] The R*-tree we used was a main memory-based version. The page size for the R*-tree was set to 1024 bytes. In our experiments, the R*-tree always fit in memory. Furthermore, for the index-based algorithm, we did not include the time to build the tree (that is, insert data points into the tree) in our measurements of execution time. Thus, our measurement for the running time of the index-based algorithm *only includes the CPU time for main memory search*. Note that this gives the index-based algorithm an advantage over the other algorithms.

[6]Our thanks to Christos Faloutsos for providing us with this code.

Partition-Based algorithm: We implemented our partition-based algorithm as described in Section 5. Thus, we used BIRCH's pre-clustering algorithm for generating partitions, the main memory R*-tree to determine the candidate partitions and the block nested-loop algorithm for computing outliers from the candidate partitions in the final step. We found that for the final step, the performance of the block nested-loop algorithm was competitive with the index-based algorithm since the previous pruning steps did a very good job in identifying the candidate partitions and their neighbors.

We configured BIRCH to provide a bounding rectangle for each cluster it generated. We used this as the MBR for the corresponding partition. We stored the MBR and number of points in each partition in an R*-tree. We used the resulting index to identify candidate and neighboring partitions. Since we needed to identify the partition to which BIRCH assigned a point, we modified BIRCH to generate this information.

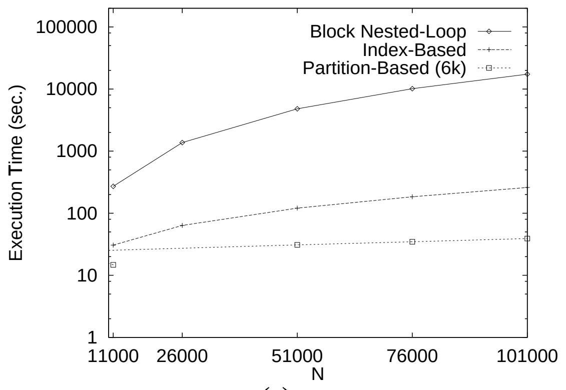
Recall from Section 5.2 that an important parameter to BIRCH is the amount of memory it is allowed to use. In the experiments, we specify this parameter in terms of the number of clusters or partitions that BIRCH is allowed to create.

6.2.2 Synthetic Data Sets

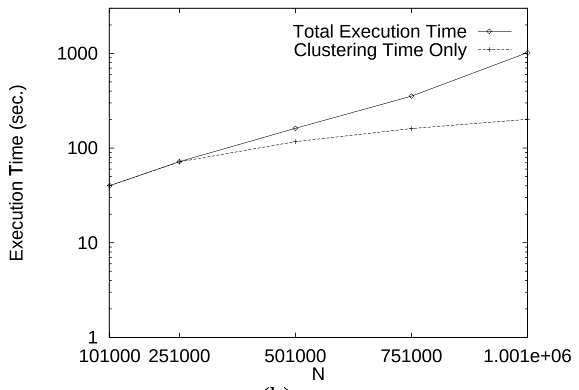
For our experiments, we used the *grid* synthetic data set that was employed in [ZRL96] to study the sensitivity of BIRCH. The data set contains 100 hyper-spherical clusters arranged as a 10×10 grid. The center of each cluster is located at $(10i, 10j)$ for $1 \leq i \leq 10$ and $1 \leq j \leq 10$. Furthermore, each cluster has a radius of 4. Data points for a cluster are uniformly distributed in the hyper-sphere that defines the cluster. We also uniformly scattered 1000 outlier points in the space spanning 0 to 110 along each dimension. Table 2 shows the parameters for the data set, along with their default values and the range of values for which we conducted experiments.

Parameter	Default Value	Range of Values
Number of Points (N)	101000	11000 to 1 million
Number of Clusters	100	
Number of Points per Cluster	1000	100 to 10000
Number of Outliers in Data Set	1000	
Number of Outliers to be Computed (n)	100	100 to 500
Number of Neighbors (k)	100	100 to 500
Number of Dimensions (δ)	2	2 to 10
Maximum number of Partitions	6000	5000 to 15000
Distance Metric	euclidean	

Table 2: Synthetic Data Parameters



(a)



(b)

Figure 6: Performance Results for N

6.2.3 Performance Results

Number of Points: To study how the three algorithms scale with dataset size, we varied the number of points per cluster from 100 to 10,000. This varies the size of the dataset from 11000 to approximately 1 million. Both n and k were set to their default values of 100. The limit on the number of partitions for the partition-based algorithm was set to 6000. The execution times for the three algorithms as N is varied from 11000 to 101000 are shown using a log scale in Figure 6(a).

As the figure illustrates, the block nested-loop algorithm is the worst performer. Since the number of computations it performs is proportional to the square of the number of points, it exhibits a quadratic dependency on the input size. The index-based algorithm is a lot better than block nested-loop, but it is still 2 to 6 times slower than the partition-based algorithm. For 101000 points, the block nested-loop algorithm takes about 5 hours to compute 100 outliers, the index-based algorithm less than 5 minutes while the partition-based algorithm takes about half a minute. In order to explain why the partition-based algorithm performs so well, we present in Table 3, the number of candidate and neighbor partitions as well as points processed in the final step. From the table, it follows that for $N = 101000$, out of the approximately 6000 initial partitions, only about 160 candidate partitions and 1500 neighbor partitions are processed in the final phase. Thus, about 75% of partitions are entirely pruned from the data set, and only about 0.25% of the points in the data set are candidates for outliers (230 out of 101000 points). This results in tremendous savings in both I/O and computation, and enables the partition-based scheme to outperform the other two algorithms by almost an order of magnitude.

In Figure 6(b), we plot the execution time of only the partition-based algorithm as the number of points is increased from 100,000 to 1 million to see how it scales for much larger data sets. We also plot the time spent by BIRCH for generating partitions— from the graph, it follows that this increases about linearly with input size. However, the overhead of the final step increases substantially as the data set size is increased. The reason for this is that since we generate the same number, 6000, of partitions even for a million points, the average number of points per partition exceeds k, which is 100. As a result, computed lower bounds for partitions are close to 0 and minDkDist, the lower bound on the D^k value for an outlier is low, too. Thus, our pruning is less effective if the data set size is increased without a corresponding increase in the number of partitions. Specifically, in order to ensure a high degree pruning, a good rule of thumb is to choose the number of partitions such that the average number of points per partition is fairly small (but not too small) compared to k. For example, $N/(k/5)$ is a good value. This makes the clusters generated by BIRCH to have an average size of $k/5$.

N	Avg. # of Points per Partition	# of Candidate Partitions	# of Neighbor Partitions	# of Candidate Points	# of Neighbor Points
11000	2.46	115	1334	130	3266
26000	4.70	131	1123	144	5256
51000	8.16	141	1088	159	8850
76000	11.73	143	963	160	11273
101000	16.39	160	1505	230	24605

Table 3: Statistics for N

k^{th} Nearest Neighbor: Figure 7(a) shows the result of increasing the value of k from 100 to 500. We considered the index-based algorithm and the partition-based algorithm with 3 different settings for the number of partitions—5000, 6000 and 15000. We did not explore the behavior of the block-nested loop algorithm because it is very slow compared to the other two algorithms. The value of n was set to 100 and the number of points in the data set was 101000. The execution times are shown using a log scale.

As the graph confirms, the performance of the partition-based algorithms do not degrade as k is increased. This is because we found that as k is increased, the number of candidate partitions decreases slightly since a larger k implies a higher value for minDkDist which results in more pruning. However, a larger k also implies more neighboring partitions for each candidate partition. These two opposing effects cancel each other to leave the performance of the partition-based algorithm relatively unchanged.

On the other hand, due to the overhead associated with finding the k nearest neighbors, the performance of the index-based algorithm suffers significantly as the value of k increases. Since the partition-based algorithms prune more than 75% of points and only 0.25% of the data set are candidates for outliers, they are generally 10 to 70 times faster than the index-based algorithm.

Also, note that as the number of partitions is increased, the performance of the partition-based algorithm becomes worse. The reason for this is that when each partition contains too few points, the cost of computing lower and upper bounds for each partition is no longer low. For instance, in case each partition contains a single point, then computing lower and upper bounds for a partition is equivalent to computing D^k for every point p in the data set and so the partition-based algorithm degenerates to the index-based algorithm. Therefore, as we increase the number of partitions, the execution time of the partition-based algorithm converges to that of the index-based algorithm.

Number of outliers: When the number of outliers n, is varied from 100 to 500 with default settings for other parameters, we found that the the execution time of all algorithms increase gradually. Due to space constraints, we do not present the graphs for these experiments in this paper. These can be found in [RRS98].

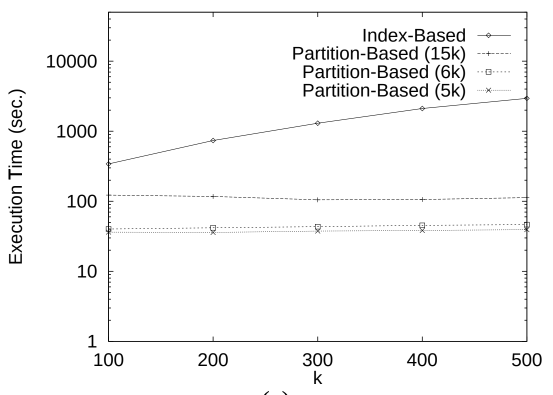
Number of Dimensions: Figure 7(b) plots the execution times of the three algorithms as the number of dimensions is increased from 2 to 10 (the remaining parameters are set to their default values). While we set the cluster radius to 4 for the 2-dimensional dataset, we reduced the radii of clusters for higher dimensions. We did this because the volume of the hyper-spheres of clusters tends to grow exponentially with dimension, and thus the points in higher dimensional space become very sparse. Therefore, we had to reduce the cluster radius to ensure that points in each cluster are relatively close compared to points in other clusters. We used radius values of 2, 1.4, 1.2 and 1.2, respectively, for dimensions from 4 to 10.

For 10 dimensions, the partition-based algorithm is about 30 times faster than the index-based algorithm and about 180 times faster than the block nested-loop algorithm. Note that this was without including the building time for the R*-tree in the index-based algorithm. The execution time of the partition-based algorithm increases sub-linearly as the dimensionality of the data set is increased. In contrast, running times for the index-based algorithm increase very rapidly due to the increased overhead of performing search in higher dimensions using the R*-tree. Thus, the partition-based algorithm scales better than the other algorithms for higher dimensions.

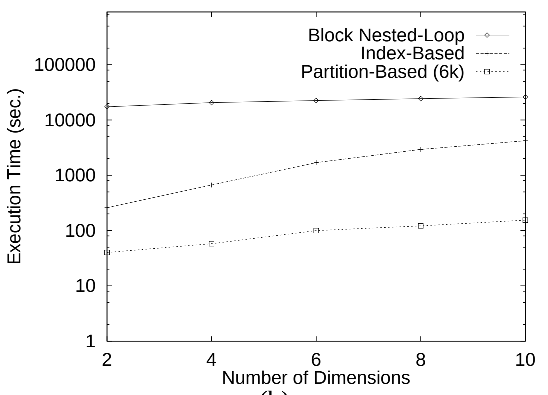
7 Conclusions

In this paper, we proposed a novel formulation for distance-based outliers that is based on the distance of a point from its k^{th} nearest neighbor. We rank each point on the basis of its distance to its k^{th} nearest neighbor and declare the top n points in this ranking to be outliers. In addition to developing relatively straightforward solutions to finding such outliers based on the classical nested-loop join and index join algorithms, we developed a highly efficient *partition-based* algorithm for mining outliers. This algorithm first partitions the input data set into disjoint subsets, and then prunes entire partitions as soon as it can be determined that they cannot contain outliers. Since people are usually interested in only a small number of outliers, our algorithm is able to determine very quickly that a significant number of the input points cannot be outliers. This results in substantial savings in computation.

We presented the results of an extensive experimental study on real-life and synthetic data sets. The results from a



(a)



(b)

Figure 7: Performance Results for k and δ

real-life NBA database highlight and reveal several expected and unexpected aspects of the database. The results from a study on synthetic data sets demonstrate that the partition-based algorithm scales well with respect to both data set size and data set dimensionality. Furthermore, it outperforms the nested-loop and index-based algorithms by more than an order of magnitude for a wide range of parameter settings.

Acknowledgments: Without the support of Seema Bansal and Yesook Shim, it would have been impossible to complete this work.

References

[AAR96] A. Arning, Rakesh Agrawal, and P. Raghavan. A linear method for deviation detection in large databases. In *Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Portland, Oregon, August 1996.

[AMS$^+$95] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. *Fast Discovery of Association Rules*, chapter 14. 1995.

[BKNS00] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jorg Sander. Lof:indetifying density-based local outliers. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 2000.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD*, pages 322–331, Atlantic City, NJ, May 1990.

[BL94] V. Barnett and T. Lewis. *Outliers in Statistical Data*. John Wiley and Sons, New York, 1994.

[EKX95] Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. A database interface for clustering in large spatial databases. In *Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Montreal, Canada, August 1995.

[GRS98] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, June 1998.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[KN98] Edwin Knorr and Raymond Ng. Algorithms for mining distance-based outliers in large datasets. In *Proc. of the VLDB Conference*, pages 392–403, New York, USA, September 1998.

[KN99] Edwin Knorr and Raymond Ng. Finding intensional knowledge of distance-based outliers. In *Proc. of the VLDB Conference*, pages 211–222, Edinburgh, UK, September 1999.

[NH94] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. In *Proc. of the VLDB Conference*, Santiago, Chile, September 1994.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of ACM SIGMOD*, pages 71–79, San Jose, CA, 1995.

[RRS98] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. Technical report, Bell Laboratories, Murray Hill, 1998.

[RS98] Rajeev Rastogi and Kyuseok Shim. Public: A decision tree classifier that integrates building and pruning. In *Proc. of the Int'l Conf. on Very Large Data Bases*, New York, 1998.

[Sam89] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[SAM98] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of olap data cubes. In *Proc. of the Sixth Int'l Conference on Extending Database Technology (EDBT)*, Valencia, Spain, March 1998.

[ZRL96] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 103–114, Montreal, Canada, June 1996.