Distributed and Parallel Databases, 17, 75–101, 2005

Optimal Data-Space Partitioning of Spatial Data for Parallel I/O

HAKAN FERHATOSMANOĞLU hakan@cse.ohio-state.edu
Computer Science and Engineering, Ohio State University, OH, USA

DIVYAKANT AGRAWAL agrawal@cs.ucsb.edu
ÖMER EĞECIOĞLU omer@cs.ucsb.edu
AMR EL ABBADI amr@cs.ucsb.edu
Computer Science, University of California Santa Barbara, CA, USA

Recommended by: Ahmed Elmagarmid

Abstract. It is desirable to design partitioning methods that minimize the I/O time incurred during query execution in spatial databases. This paper explores optimal partitioning for two-dimensional data for a class of queries and develops multi-disk allocation techniques that maximize the degree of I/O parallelism obtained in each case. We show that hexagonal partitioning has optimal I/O performance for circular queries among all partitioning methods that use convex non-overlapping regions. An analysis and extension of this result to all possible partitioning techniques is also given. For rectangular queries, we show that hexagonal partitioning has overall better I/O performance for a general class of range queries, except for rectilinear queries, in which case rectangular grid partitioning is superior. By using current algorithms for rectangular grid partitioning, parallel storage and retrieval algorithms for hexagonal partitioning can be constructed. Some of these results carry over to circular partitioning of the data—which is an example of a non-convex region.

Keywords: data-space partitioning, two-dimensional data, parallel I/O, disk and page allocation, range query

1. Introduction

Spatial databases and Geographical Information Systems (GIS) [9] have gained added importance as a result of recent developments in information technology. In GIS applications, the data objects are represented as two-dimensional vectors, and the similarity between objects are defined by a distance function between corresponding vectors. Several index structures have been proposed for the retrieval of spatial data [2, 17, 18, 24, 28, 29]. The common approach is to group the objects according to their spatial locations, partition the data space, and store the created groups as pages in physical storage.

Most common indexing algorithms for spatial data are based on clustering the data points using a rectangular organization [2, 17, 18, 29]. Grid based file structures have been effectively used to index spatial data [24]. There have also been several approaches based on grid partitioning [17, 29]. Rectangles are the most common geometrical shape that have been used to index spatial data, although other shapes have also been proposed. For example, circles are suitable for multi-dimensional indexing for similarity searching [32]. One

difficulty with using circles is that a true partitioning of the data space in the mathematical sense is often not possible (i.e., we need to allow for overlaps). These overlaps cause degradation in the query performance, as a large number of intersections directly increases the I/O cost of query processing.

Because of their simplicity in hashing and mapping to physical storage, *regular* equi-sized partitioning such as regular grid partitioning is widely used for retrieval and storage of spatial data. An important application of this type of partitioning is multi-disk *declustering* in which the rectangles (buckets) are allocated to multiple I/O devices in such away that neighboring rectangles are allocated to distinct disks as much as possible. In this way, locality of a spatial query results in parallel retrieval of the buckets involved in processing the query. Numerous methods have been proposed for declustering such as Disk Modulo (DM) [10], Fieldwise Exclusive OR (FX) [23], Hilbert (HCAM) [12], Near Optimal Declustering (NoD) [3], General Multidimensional Data Allocation (GMDA) [22], Cyclic Allocation Schemes [26, 27], Coloring [1], Golden Ratio Sequences [7], Hierarchical [6], Discrepancy [8], and replicated declustering [30]. These approaches are all based on *non-overlapping* rectangular partitions. Another successful application of regular partitioning is the vector approximation (VA) based indexing for multi-dimensional data [16, 31]. The VA-file approach divides the data space into 2^b non-overlapping rectangles where b is specified by the user. Each partition is labeled by a b-bit binary string which is used to represent the data points that fall into the partition.

Evidently, minimizing the number of I/O operations in query processing is crucial for fast response times. This cost can be reduced by reducing the expected number of page retrievals during the execution of queries. The expected number of page retrievals directly depends on the underlying technique that is used to organize the data set, i.e., partitioning of the data [5, 11, 13, 14]. It is therefore important to develop partitioning techniques which minimize the expected number of partitions retrieved by a query. Since rectangles are effective for non-overlapping partitioning, a natural question is whether rectangular grid based partitioning is the optimal approach for non-overlapping partitioning to minimize the I/O cost.

In this paper, we explore optimal partitioning for various types of queries. In particular, we show that *hexagonal* partitioning has optimal I/O performance for circular queries compared to all possible non-overlapping partitioning techniques that use convex regions. For rectangular queries, except for the case of rectilinear queries for which rectangular grid partitioning gives superior performance, hexagonal partitioning has overall better I/O cost for a general class of range queries. We also develop simple methods for the storage and retrieval of hexagonal partitioning in a multi-disk setting by applying the already developed algorithms for rectangular grid partitioning. Although partitioning schemes that use convex regions cover a wide range of applications by nature of their simplicity and efficiency, we extend our results for the class of partitioning techniques that use non-convex regions as well, thus demonstrating the general applicability and usefulness of our approach.

This paper is organized as follows: in Section 2 we discuss the importance of data organization for minimizing the I/O cost of spatial queries and summarize available partitioning techniques. In Section 3, we analyze these techniques with respect to the I/O cost of spatial queries and identify the optimal partitioning technique for each query type.

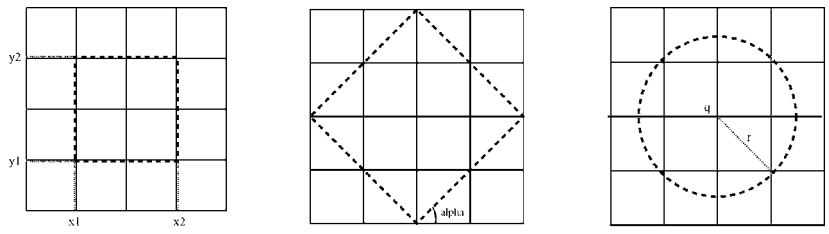
Section 4 discusses the storage and retrieval of hexagonal partitioning using rectangular grids. Section 6 discusses the relationship between the partitioning techniques and the I/O cost in a more general setting, where the partitions are allowed to be non-convex. Conclusions are presented in Section 7.

2. Data organization for efficient spatial queries

The two most common spatial queries are *range queries* and *similarity queries*. In a range query, the user specifies an area of interest and all data points in this area are retrieved. In a similarity query, a query point is specified and all points "similar" to the query point are retrieved. A popular related query is *k-nearest neighbor* (*kNN*), which is the determination of the k closest points to the query point q. Traditional range queries have been specified using *rectilinear queries*, i.e., rectangular regions with sides parallel to coordinate axes. A rectilinear range query $Q_r = ([a_1, b_1], [a_2, b_2])$ specifies a range of values for each dimension. The result of the query is the set of all data objects that have values within the specified ranges. More generally, a rectangular query can be defined as a rectilinear rectangle with a rotation angle of α with respect to the x-axis. For example, a rectilinear square range query with a rotation of $\pi/4$ gives us a *diamond query* (see figure 1(b)). Another commonly used query is *circular range query* $Q_c = (q, r)$, also referred to as an ϵ-similarity query. It specifies a query point q and a radius $r = \epsilon$, which together define the acceptable region of similarity. All data objects that fall into the circle defined by the pair (q, r) are in the answer set.

It is interesting to note that range queries of different shapes correspond to similarity queries under different metrics. For example, the rectilinear rectangular range query corresponds to similarity in the L_∞ metric, the diamond range query to similarity in the L_1 metric, and the the circular range query to the L_2 metric.

In both range and similarity queries, the pages that potentially contain a portion of the query result are retrieved. As mentioned before, the spatial data objects are grouped according to their spatial locations and are stored as pages in physical storage. An ideal technique would require $\lceil \frac{QueryResultSize}{PageCapacity} \rceil$ number of page accesses to answer any query [21]. However, this is not in general possible. A careful investigation is needed to develop a



(a) Rectilinear Query (b) Diamond Query (c) Circular Query

Figure 1. Range queries intersecting all partitions.

technique that is close to the ideal case, i.e., maximally reduces the I/O cost on the average. For a uniformly distributed data space of unit area, the best possible (optimal) cost for a range query with area A is equal to $\lceil Ap \rceil$, where p is the total number of pages. If a query is executed in two differently organized copies of the same data set, different numbers of pages will be retrieved depending on the way the data-space is initially organized. For efficient queries, the underlying organization of the data must be designed to reduce the expected number of pages retrieved by the queries. An organization of a database is I/O *optimal* if it minimizes the expected number of page accesses incurred by the queries.

Regular equi-sized partitioning is widely used for the organization of spatial data since it provides efficient functions for hashing and mapping to pages in storage. The general approach is first to tile the data space into disjoint regular rectangular regions and then to map each partition to a physical page. It is known that it is not possible to tile the space with regular non-overlapping identical convex polygons with more than 6 edges. Furthermore there are only three basic tile shapes (triangles, rectangles, and hexagons) for regular partitioning [25]. We consider these three basic shapes for partitioning and compare their behavior under various conditions. Equilateral triangles, squares, and regular hexagons are considered for triangles, rectangles, and hexagons respectively. All of the techniques discussed in this paper have simple hashing properties that can be used to map the partitions to physical storage easily.

3. I/O cost for various query types

Consider different partitioning techniques that minimize the expected number of pages retrieved for a given query. The partitions that have to be accessed are the ones that intersect the range query region. We compute the expected number of these by using the idea of *Minkowski sum* [4, 31]. The Minkowski sum of two planar regions R and S is defined as the vector sum of these sets in a vector space, i.e., $M(R, S) = \{x + y \mid x \in R, y \in S\}$. This operation essentially expands one region by the other. As an example, the Minkowski sum of a square page of side length c and a circular region with radius r is the enlarged object $M(square_c, circle_r)$, which is obtained by moving the center of the circular query over the surface of the square (figure 2b). It consists of all points that are either in the square page or within distance r to the boundary of the page. The points within this enlarged object correspond to the center of all possible circular queries that intersect this page. Assuming uniform distribution, the fraction of the area of the enlarged object to the area of the data space is the probability $P(page)$ that the corresponding page is being accessed. When the data space is the unit square $[0, 1]^2$, the area of the Minkowski sum gives the probability that the page is accessed. Note that the regions created by the Minkowski sum of the partitions on the boundary of the two-dimensional data space are negligible when compared to the union of regions created by Minkowski sums of all partitions. The expected number $E(q)$ of pages which are intersected by a query q is the sum of the probabilities of the pages intersected by the query. For regular equi-sized partitioning, i.e., p partitions of identical shape, the expected number of intersected partitions by the query is p times the probability of a page being accessed, i.e., $E(q) = p \cdot P(page)$. Therefore, for a partitioning scheme that tiles the space with pages of shape s, the expected numbers of intersected partitions for

a query q of any shape is

$$E_s(q) = p \cdot M(s, q), \tag{1}$$

where p is the number of partitions. For example, for a square partitioning

$$E_{\text{square}}(q) = p \cdot M(square_c, q).$$

We next investigate which type of partitioning must be used to minimize the I/O cost, assuming uniform distribution. We consider circular, rectilinear and rotated square queries, which correspond to similarity using L_2, L_∞, and L_1 metrics respectively.

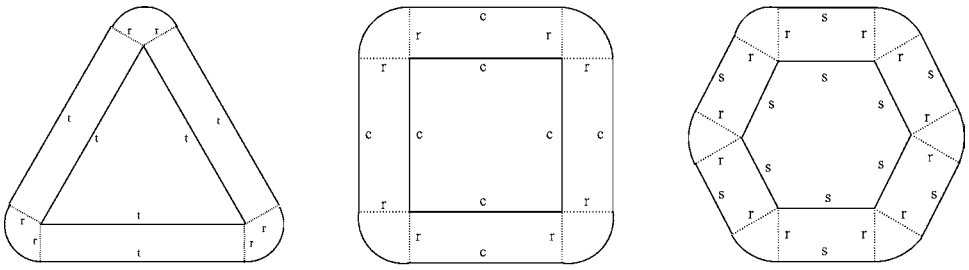
3.1. *I/O cost for circular range queries*

To analyze the performance for circular range queries, we first compute the Minkowski sums of triangle, square, and hexagon shapes with respect to a circular region of radius r. Figure 2 shows the Minkowski sum regions for these three cases, i.e., $M(triangle_t, circle_r)$, $M(square_c, circle_r)$, and $M(hexagon_s, circle_r)$, respectively. The values of each edge, t, c, and s, are assigned such that the area of each triangle, square, and hexagon partition is equal to $1/p$, i.e., $\frac{t^2\sqrt{3}}{4} = c^2 = s^2\frac{3\sqrt{3}}{2} = 1/p$. We obtain the following expressions for the areas of the Minkowski sums:

$$M(triangle_t, circle_r) = \frac{t^2\sqrt{3}}{4} + 3tr + \pi r^2 \tag{2}$$

$$M(square_c, circle_r) = c^2 + 4cr + \pi r^2 \tag{3}$$

$$M(hexagon_s, circle_r) = s^2\frac{3\sqrt{3}}{2} + 6sr + \pi r^2 \tag{4}$$



(a) M(Triangle,Circle) (b) M(Square,Circle) (c) M(Hexagon,Circle)

Figure 2. Minkowski sums (M) w.r.t. circles.

Note that the first term in each equation corresponds to the area of each partition, i.e., $\frac{t^2\sqrt{3}}{4} = c^2 = s^2\frac{3\sqrt{3}}{2} = 1/p$. From these equations, $t = s\sqrt{6}$, and $c = s\sqrt{\frac{3\sqrt{3}}{2}}$, therefore for $\sqrt{\frac{3\sqrt{3}}{2}} \approx 1.612$, and $\sqrt{6} \approx 2.45$, Eqs. (2) and (3) becomes:

$$M(triangle_t, circle_r) \approx s^2\frac{3\sqrt{3}}{2} + 7.35sr + \pi r^2. \tag{5}$$

$$M(square_c, circle_r) \approx s^2\frac{3\sqrt{3}}{2} + 6.45sr + \pi r^2. \tag{6}$$

Comparing (4), (5) and (6), we find

$$M(hexagon_s, circle_r) < M(square_c, circle_r) < M(triangle_t, circle_r).$$

This means that for circular regions the Minkowski sum of a hexagon is always less than the Minkowski sum of a square and the Minkowski sum of a triangle. Substituting this result into Eq. (1), we finally find

$$E_{\text{hexagon}}(circle_r) < E_{\text{square}}(circle_r) < E_{\text{triangle}}(circle_r).$$

Thus the expected number of partitions intersected by a circular query in a hexagonal grid is always less than the expected number in a square grid and a triangular grid. Therefore if the sole or dominant types of the queries in a spatial database have circular shapes, then hexagonal partitioning of the data space will give better results in terms of the number of page accesses.

We now generalize the superiority of hexagonal partitioning further and prove that it is optimal among all non-overlapping partitioning techniques of equal area convex regions, with respect to the number of partitions retrieved as a result of a circular query of any size.

Lemma 1. *For a convex polygon with a perimeter of length M, the Minkowski Sum with respect to circular regions of radius r is*:

$$M(polygon, circle_r) = area(polygon) + M \cdot r + \pi r^2 \tag{7}$$

Proof: For a convex polygon of n sides, the Minkowski sum with a circle of radius r contains the polygon itself, plus n rectangles with sides r and the corresponding side of the polygon, plus n pies with an angle of Γ_i, for $1 \leq i \leq n$. The area of the n rectangles sum up to $M \cdot r$, where M is the perimeter of the polygon. Each Γ_i is equal to $\pi - \Theta_i$ radians, where Θ_i is the corresponding angle in the polygon. Since the summation of all these n angles of pies is 2π, $\sum_1^n(\pi - \Theta_i) = \pi n - \pi(n-2) = 2\pi$, they sum up to a full circle. □

Lemma 2. *Minimizing the perimeter of the shape that is used for partitioning also minimizes the expected number of partitions intersecting the circular query.*

Proof: In Eq. (7) r does not change with partitioning, and $area(polygon) = 1/p$. The only parameter that makes a difference is perimeter M that is used in the pages. This is true for all possible r values that a circular query can take. □

Minimizing the perimeter of each partition minimizes the total boundary of the partitioning, which is defined as the length of the boundaries that are used to divide the data-space into partitions (see Appendix A for an example). If each non-overlapping partition uses smaller perimeters to cover larger areas, the total lengths of boundaries of these partitions are also minimized (perimeter comparison of the three basic shapes is given in Appendix B). It is known that any partitioning of the plane into regions of equal area has perimeter at least that of the regular hexagonal honeycomb tiling [19]. The surprising instance of the geometry of the beehives is seen as the best that could be done for their major purpose. In 1743, Colin Mac Laurin summarizes this isoperimetric property of the beehives as follows: "The geometry of the beehive supports *least wax* for containing *the same quantity of honey*, and which has at the same time a very remarkable regularity and beauty, connected of necessity with its frugality" [20].

By Lemmas 1 and 2, partitioning that minimizes the total boundary also minimizes the expected number of partitions retrieved as a result of a circular query. Since hexagon partitioning minimizes the total boundary among all possible equal area partitioning methods, it also minimizes the expected number of partitions retrieved as a result of a circular query. Hence, we have the following result:

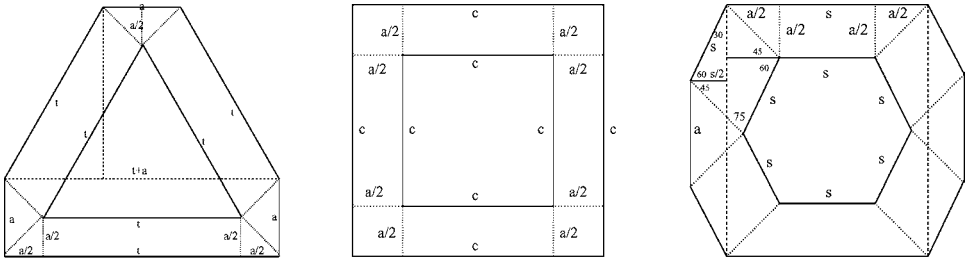
Theorem 1. *Hexagonal partitioning is I/O optimal for circular queries among all non-overlapping partitioning techniques using equal area convex regions.*

3.2. I/O cost for square range queries

Now we consider the I/O cost for square range queries. We start with rectilinear square query analysis on the three basic partitioning discussed in this paper. Then, we analyze a more general class of square range queries where the sides of the range query do not have to be parallel to the axes. We establish that hexagonal partitioning has better average performance than a square grid for the general class of square range queries.

3.2.1. Rectilinear square range queries. For the analysis of rectilinear square range queries, we first compute the Minkowski sums of the three shapes and a rectilinear square region (see Appendix C for a similar analysis of rectilinear rectangular range queries). Figure 3 illustrates the Minkowski sum for each case. As can be seen from the figures, we have $M(square_c, rectil2near_a) = (c+a)^2$. Substituting $c = s\sqrt{\frac{3\sqrt{3}}{2}}$, we find the Minkowski Sum of the square page as follows:

$$M(square_c, rectilinear_a) = \frac{3\sqrt{3}}{2}s^2 + 2\sqrt{\frac{3\sqrt{3}}{2}}as + a^2 \tag{8}$$



(a) M(Triangle,Rectilinear Square)

(b) M(Square,Rectilinear Square)

(c) M(Hexagon,Rectilinear Square)

Figure 3. Minkowski sums w.r.t. rectilinear squares.

The total area of the $M(hexagon_s, rectilinear_a)$ is the area of the rectangle inside the M plus the two trapezoids created on both sides of M. Therefore,

$$M(hexagon_s, rectilinear_a) = (s + a)(s\sqrt{3} + a) + \frac{(2a + s\sqrt{3})s}{2} \tag{9}$$

$$M(hexagon_s, rectilinear_a) = \frac{3\sqrt{3}}{2}s^2 + (2 + \sqrt{3})as + a^2 \tag{10}$$

Similarly,

$$M(triangle_t, rectilinear_a) = \frac{3\sqrt{3}}{2}s^2 + (\frac{3\sqrt{2}}{2} + \sqrt{6})as + a^2 \tag{11}$$

Comparing (8), (10) and (11), we have

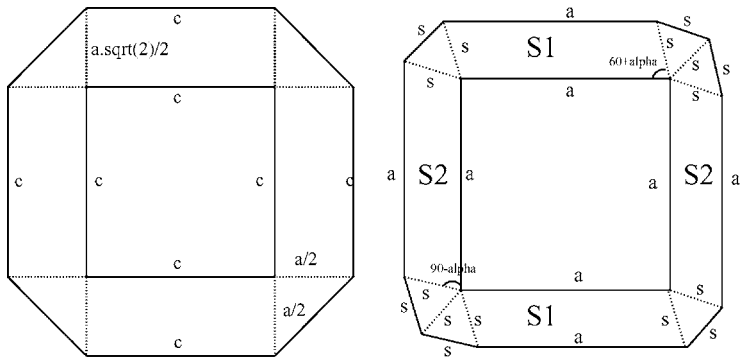
$$M(square_c, rectilinear_a) < M(hexagon_s, rectilinear_a) < M(triangle_t, rectilinear_a).$$

Therefore

$$E_{\text{square}}(rectilinear_a) < E_{\text{hexagon}}(\text{rectilinear}_a) < E_{\text{triangle}}(rectilinear_a).$$

We conclude that for rectilinear square queries, the expected number of intersected partitions in a square grid is less than the one in a hexagonal grid and a triangular grid.

3.2.2. General square range queries. In the previous section, we focused on rectilinear square range queries, but queries can take any orientation and are not restricted to have sides parallel to the axes, e.g., as in diamond queries. In this section, we analyze square queries with *arbitrary orientation*. We will compare the hexagonal and square partitioning by computing the Minkowski Sums for square and hexagon tiles with a rotated square region. The square region has an angle α with the x-axis. Special cases include the diamond query,



(a) M(Square,Square) (b) M(Square,Hexagon) (= M(Hexagon,Square))

Figure 4. Minkowski sums w.r.t. squares (with orientation α).

i.e., $\alpha = \pi/4$, and the rectilinear square query, i.e., $\alpha = 0$. In this section, for simplicity, we assume that the area of each partition is 1, i.e., $c^2 = s^2 \frac{3\sqrt{3}}{2} = 1$. Figure 4 illustrates the case when $\alpha = \pi/4$. Because of the symmetric property of M, i.e., $M(hexagon_s, square_a) = M(square_a, hexagon_s)$, for simplicity we illustrate $M(square_a, hexagon_s)$ in figure 4.

The sum of a square page with respect to a square query with an angle α with the x-axis is:

$$M(square_1, square_a) = 1 + a^2 + 2\sqrt{2}a \cdot \sin(\pi/4 + \alpha) \tag{12}$$

where $0 \leq \alpha \leq \frac{\pi}{4}$.

Similarly, the M of a hexagon with respect to a square with a rotation angle of α, is computed as $M(hexagon, square_a) = a^2 + 1 + 2(S_1 + S_2)$, where $S_1 = as \cdot \sin(\frac{\pi}{3} + \alpha)$ and $S_2 = as \cdot \sin(\frac{\pi}{2} - \alpha)$. Therefore,

$$M(hexagon, square_a) = a^2 + 1 + 2as\left(\sin\left[\frac{\pi}{3} + \alpha\right] + \sin\left[\frac{\pi}{2} - \alpha\right]\right) \tag{13}$$

where $0 \leq \alpha \leq \frac{\pi}{6}$.

The reason that the rotation angle α varies between 0 and $\pi/6$ in the hexagon and between 0 and $\pi/4$ in the square is that the symmetry is captured within these angles. There is no need to compute other angles, because rotating an angle of β not in this range gives the same result as rotating an angle of $\alpha = \beta \bmod \pi/6$ for a hexagon and $\alpha = \beta \bmod \pi/4$ for a square. This behavior of the rotation can also be seen by the periodicity in figure 5. Figure 5 shows the average number of page accesses for various square queries with varying angles. The $x-$axis of the figure corresponds to the angle α in terms of radians. For example, an angle of $\pi/4$ (≈ 0.785) corresponds to the diamond query which is the shape for similarity queries in the L_1 metric. For most angles which correspond to different query types, including

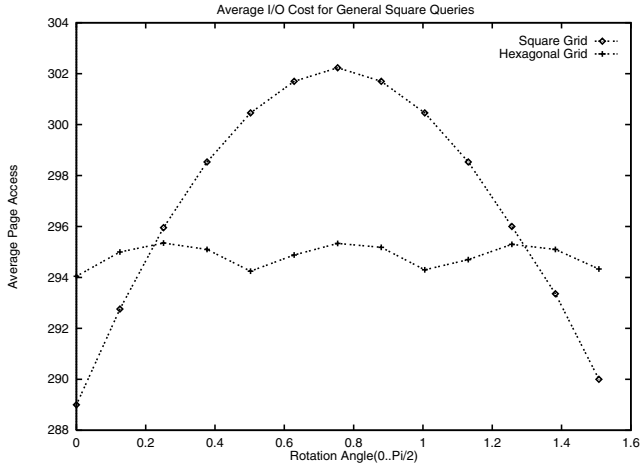


Figure 5. Average I/O cost for general square queries with varying angles.

the diamond query, hexagonal partitioning achieves better performance results. Since we have the general formula for the square query specified by a center and an angle, we can compute the expected number of partitions intersected by such queries for both square and hexagonal grids. By integrating over all such possible queries and computing the expected number by taking the uniform average, we compute the expected M, E_M, of each technique. From Eqs. (12) and (13) for all possible values of α, we compute the expected M for each partition.

$$E_{M_{\text{square}}}(square_a) = \frac{4}{\pi}\int_{\pi/4}^{\pi/2}(M(square_1, square_a)$$
$$= a^2 + 1 + \frac{8\sqrt{2}a}{\pi}\int_{\pi/4}^{\pi/2}\sin\alpha d\alpha$$

Therefore,

$$E_{M_{\text{square}}}(square_a) = 1 + a^2 + \frac{8a}{\pi} \tag{14}$$

Similarly, for hexagons,

$$E_{M_{\text{hexagon}}}(square_a) = 1 + a^2 + \frac{2as\cdot 6}{\pi}\int_0^{\pi/6}[\sin(\pi/3+\alpha) + \sin(\pi/4-\alpha)]\,d\alpha$$

Solving this equation, we finally have,

$$E_{M_{\text{hexagon}}}(square_a) = 1 + a^2 + \frac{12as}{\pi} \approx 1 + a^2 + \frac{7.44a}{\pi} \tag{15}$$

where $s \approx 0.62$ since we assumed the area of the hexagon is 1, i.e., $s^2\frac{3\sqrt{3}}{2} = 1$.

Comparing (14) and (15), we find $E_{M_{\text{hexagon}}}(square_a) < E_{M_{\text{square}}}(square_a)$. Hence, we conclude that hexagonal partitioning minimizes the expected number of partitions intersected by a square query with any orientation α, compared to the square grid partitioning. The analysis can be easily extended for rectangular queries.

4. Retrieval and storage of hexagonal partitioning

We have shown that hexagonal partitioning is effective for range queries, and hence can be used as an effective alternative for regular grid partitioning. Traditional retrieval methods developed for single disk and single processor environments may be ineffective for the storage and retrieval of spatial data in multiprocessor and multiple disk environments. It is essential to develop methods that are optimized for such environments. In this section, we develop a multi-disk organization of the data using declustering (distributing the neighboring partitions across multiple disks) and clustering (storing spatially close partitions close in disk) of hexagonal partitioning.

We start by defining a hash function for regular hexagonal partitioning. The hashing function finds the corresponding hexagonal partition of a given data point and is needed for both declustering and clustering purposes. Hash functions for rectangular partitioning are very simple and well-known. We will use rectangular hashing in the development of the hexagonal hashing. We divide the data space logically into a regular grid of rectangles with sides (s, h), where s is the side length of the hexagonal partitions and $h = \frac{s\sqrt{3}}{2}$ (see figure 6). A hexagonal partition is defined by $H(i, j)$, where i is the row number and j is the column number of the partition. Similarly, a (logical) rectangular partition is defined by $G(i, j)$.

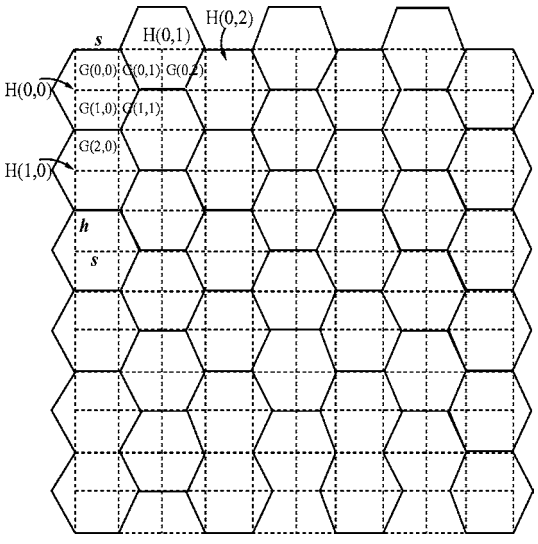


Figure 6. Hexagonal hashing using rectangular grid.

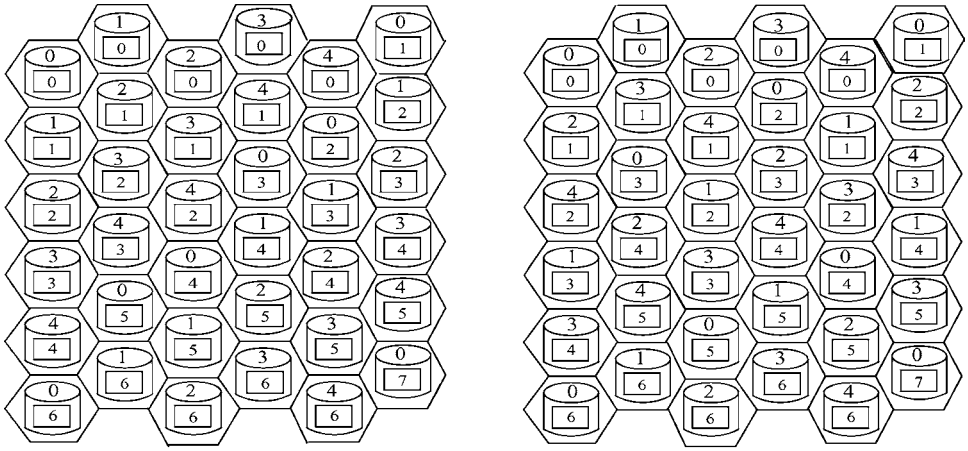
$$
\begin{array}{|l|}
\hline
G(2m, 3n) \subseteq H(m, 2n) \\
G(2m, 3n+1) \subseteq H(m, 2n) \cup H(m, 2n+1) \\
G(2m, 3n+2) \subseteq H(m, 2n+1) \cup H(m, 2n+2) \\
G(2m+1, 3n) \subseteq H(m, 2n) \\
G(2m+1, 3n+1) \subseteq H(m, 2n) \cup H(m+1, 2n+1) \\
G(2m+1, 3n+2) \subseteq H(m+1, 2n+1) \cup H(m, 2n+2) \\
\hline
\end{array}
$$

Figure 7. Mapping of rectangular grid partitions to hexagonal partitions.

Obviously, the number of rectangular partitions is more than the number of hexagonal partitions. Depending on the location, some of the partitions in this regular grid fall entirely into a single hexagonal partition, and some fall into two hexagonal partitions. For example, in figure 6, $G(0, 0)$ is entirely in $H(0, 0)$. Therefore, if a point is found to be in $G(0, 0)$, it is also in $H(0, 0)$. On the other hand $G(1, 1)$ falls mostly in $H(1, 1)$ but also in $H(0, 0)$. Each grid partition can be mapped to one or two hexagonal partitions. Therefore, given a data point we can hash the point using hashing on a regular grid and find the corresponding hexagon(s). If there are two hexagons, an additional simple check (whether the point is in the first hexagon) is needed to identify the hexagon where the point is located. The different cases for mapping of rectangular grid partition to hexagonal partitions are shown in figure 7. A grid partition $G(i, j)$ is mapped to hexagonal partitions with a simple analysis on the value of (i mod 2) and (j mod 3). For example, grid partition $G(2m, 3n)$, i.e., $i \bmod 2 = 0$ and $j \bmod 3 = 0$, is mapped to $H(m, 2n)$. Given a point in $G(2m, 3n)$, e.g., $m = 2$ and $n = 1$ so $G(4, 3)$, it is located only in $H(m, 2n)$ (see figure 7), e.g., $H(2, 2)$.

We now discuss how to decluster the hexagonal partitions over multiple devices and cluster them in each disk. An observation important for two-dimensional declustering is the following. For regular grid partitioning, a bucket has 4 direct neighbors each of which shares an edge with the bucket, and 4 indirect neighbors each of which shares a point with the bucket. For regular hexagonal partitioning, there are no indirect neighbors but only 6 direct neighbors which are equally important for the declustering function. Since the importance of the neighbors is equal, the task of declustering in hexagonal partitioning is well-defined. A metric, called *counts*, for the quality of neighbor declustering is defined and used in [3, 27]. It is a measure of how far the allocation is from being optimal in terms of declustering of neighbors of some degree. The *count* for an allocation scheme is the sum of count for each bucket. The count for a bucket is the number of neighbors of the bucket that are allocated to the same disk as the bucket itself. Thus a lower count indicates better declustering neighbors. A count of 0 indicates optimal neighboring declustering.

Figure 8(a) illustrates a two-dimensional data space that is partitioned into 36 regular hexagonal partitions with 6 rows and 6 columns. The labels in the figure specify the disks as well as the page assignments of each partition on that disk, assuming that there are five disks in the system. The ovals have the disk numbers, the rectangles have the page numbers. The disk allocation is needed for declustering and the page allocation is needed for clustering purposes. The disk assignment in figure 8(a) is developed by applying the Disk Modulo



(a) DM declustering on hexagons

(b) Cyclic declustering on hexagons

Figure 8. Declustering of hexagonal partitions.

(DM) declustering [10] for hexagonal partitioning. We assign the pages in row major order for each disk. In general, the physical location of a partition, $PartitionId = (i, j)$, where i is the row number and j is the column number, can be computed as follows:

$$disk(i, j) = (i + j) \bmod M$$
$$page(i, j) = \left\lfloor \frac{iN + j}{M} \right\rfloor$$

where N is the number of partitions in each dimension and M is the number of disks.

More effective but complex disk allocation techniques have been developed for regular grid partitioning [22, 26]. As an example, figure 8(b) illustrates the Cyclic Allocation Technique [13, 26] applied to hexagonal partitioning. Partition (0, 0) is assigned to device 0. Next, the partitions along row 0 are assigned to consecutive devices, i.e., partition $(0, j)$ is assigned to device $j \bmod M$. Each partition in row 1, $(1, j)$ is assigned to device $(H + j) \bmod M$, H devices away from the device on which the partition from the same column in row 0 was assigned. Similarly, each partition on row i, (i, j) is assigned to device $(Hi + j) \bmod M$. H is called the skip value and best H can be chosen according to the number of partitions. In figure 8(b), the skip value in each row is chosen to be 2. The page allocation function for Cyclic Allocation [13] can be applied to hexagonal partitioning in the same way as disk allocation. Note that in this example, each hexagonal partition is assigned to a different device from all its 6 neighbors, i.e., no two neighbor partitions are allocated to the same device, which was not the case with DM declustering in figure 8(a). With enough devices available, Cyclic Allocation on a rectangular grid guarantees that no two neighbors (both direct and indirect neighbors) of a bucket are allocated to the same device [27].

In a hexagonal partitioning, the neighbors of a bucket (i, j) are the buckets $\{(i-1, j), (i, j-1), (i, j+1), (i+1, j), (i+1, j-1), (i+1, j+1)\}$. The first four of these neighbors correspond to the direct neighbors in a rectangular grid and the last two correspond to the two of the indirect neighbors. Since the neighbor set in a hexagonal grid is a subset of the neighbor set in a rectangular grid, Cyclic Allocation on a hexagonal grid also guarantees that no two neighbors (which are all equally important) are allocated to the same device. In this case, since the number of neighbors considered for declustering in a rectangular grid is greater than the one in a hexagonal partitioning, the number of devices needed for optimal declustering for a rectangular grid is also larger than hexagonal partitioning.

5. Experimental analysis over real data

The hexagonal partitioning clearly performs better for the case of uniform partitioning, which has been the major focus of the declustering techniques proposed in the literature so far. A representative result for the uniform data analysis can be seen at figure 5. For the completeness, we also performed experiments on real-life data.

The real-data experiments in this section have been carried out on two data sets: North East (NE) and Sequoia. The NE data set contains 123,593 postal addresses, which represent three metropolitan areas (New York, Philadelphia and Boston), hence three clusters, with a lot of noise, in the form of uniformly distributed rural areas and smaller population centers. The Sequoia dataset contains the locations of 62,556 Californian Giant Sequoia groves. The experiments are run on various types of queries: circular, rectilinear and general rectangular with varying rotation angle α, and diamond. The results are reported for the average number of pages retrieved for randomly chosen 100 query points and for various number of partitions over the data space.

For circular queries, we varied the radius of the query $q = \{p(x, y) : d(p, q) < R\}$. The values are for values of R from 0.09–0.14 in intervals of 0.1, for hexagonal, square and triangular partitions of equal sizes, the number of partitions ranging from 600 to 1800. For rectangular queries, we did three experiments: rectilinear rectangles, diamond queries, and rotated rectangular queries with varying rotation angle α. The rectilinear rectangular query corresponds to $\alpha = 0$ and the diamond query corresponds to $\alpha = \pi/4$. Another parameter is the side length of the queries, A which is varied from 0.09 to 0.14 in increments of 0.01.

For all the experiments the hexagonal partitioning consistently performed better than the traditional grid partitioning. The speedups are naturally not very huge but they are significant, and they consistently show that hexagonal partitioning is a better alternative to the grid partitioning. Here we illustrate some representative results for the NE data set. The results for Sequoia data set are comparable to or slightly better than the NE results. For instance, for circular queries of all radius values on the NE data set with 1600 partitions, the average number of pages accesses is 3pages and 5diamond query of size $A = 1.1$ over 1100 partitions, the average number of pages accessed for hexagonal partitioning is 28, and it is 31 for the square grid partitioning. The relative numbers are similar for other parameters that are mentioned above.

The figure 9 illustrates the average number of pages retrieved by general square range queries with varying angles. It can be seen as real-data analogue of figure 5. This particular

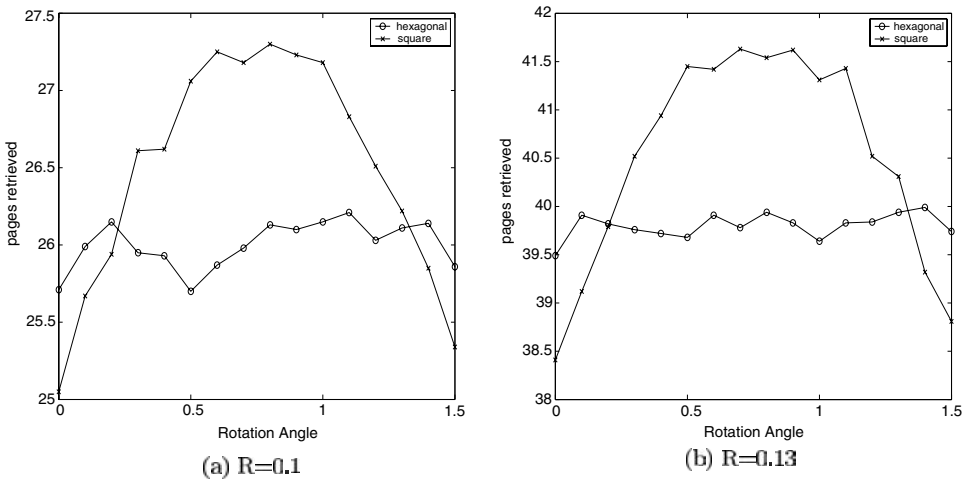


Figure 9. Average I/O cost for general square queries varying angles on NorthEast data.

figure presents results for randomly chosen 100 queries with side of 0.1 and with side of 0.13 over 1600 partitions. The behavior of the two partitioning techniques for various angles is consistent with the theoretical analysis performed in the previous section. The larger query accesses naturally more pages.

6. Partitioning using non-convex regions

The optimality of hexagonal partitioning is proved among all possible partitioning techniques using convex regions. This proof was based on the observation that the total boundary used for partitioning is the only factor to minimize the cost for ϵ-range queries. Although partitioning using convex regions covers a large spectrum of applications, in this section, we will generalize this result for a general class of partitioning techniques which also includes non-convex regions. In particular, we will show that the expected number of regions a circular query of radius ρ intersects is minimized when the total boundary of the regions (of any kind) forming the partition is minimized. However, although this is a more general result including also the non-convex regions, the optimality result in this section is valid for small size queries. Although there is no proof for theoretical optimality for large queries, our experiments on large queries presented in this section are practically in perfect agreement with the general optimality result.

6.1. The total boundary length and I/O cost (pages accessed)

We will show that the expected number of partitions a small query of radius ρ intersects is minimized when the total boundary of all the partitions forming the grid is minimized. This is not exactly right if ρ is not small, since the number of partitions meeting at a point is

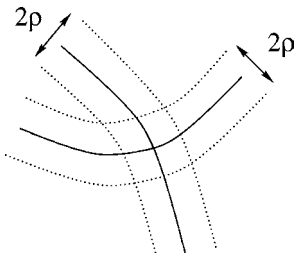


Figure 10. Crossing partition boundaries.

important in the calculation of the expected value, whereas the length of the boundary does not directly take this into account (i.e. it treats the number of partitions intersected as either 1 (one side of a boundary line) or 2 (intersecting a boundary line)). However for small ρ, we argue that this assumption is valid as follows:

Consider a strip of width 2ρ around each boundary line, with the boundary line running at the center as shown in figure 10. Any query circle of radius ρ centered outside region S in the unit circle formed by these strips is contained in a single partition. If the total area of S is A, then the expected number E of partitions intersected by a random query circle of radius ρ satisfies

$$\frac{1}{\pi}(\pi - A) + 2\frac{A}{\pi} \leq E \leq \frac{1}{\pi}(\pi - A) + n\frac{A}{\pi}.$$

This is because each query circle with center in S intersects at least 2 and at most n partitions. Thus

$$1 + \frac{A}{\pi} \leq E \leq 1 + (n-1)\frac{A}{\pi}, \tag{16}$$

and for any fixed n, E approaches its minimum possible value of 1 as A approaches 0. Therefore for small enough ρ, the expected value E is minimized for a partitioning into equal tiles with minimum boundary. This is a type of an isoperimetric problem. Note that in (16), it is possible to use planarity and reduce the quantity $n - 1$ for boundaries that are not pathological. This is because by Euler's formula the average degree of a vertex in a planar graph is no larger than 6.

6.2. Analysis on circular data space partitioning

As an example of a class of non-convex partitioning we consider circular data-space partitioning. The overall region of the data-space is also taken to be a circle. One extreme possibility is to partition the data-space by concentric circles (rightmost partition in figure 11). Another extreme way is to partition the data-space into wedges without using any concentric

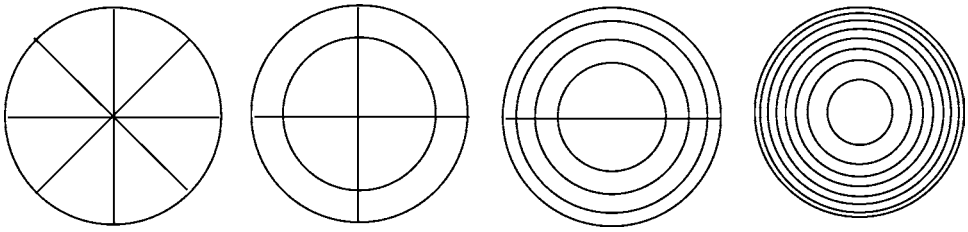


Figure 11. Equi-area subdivision of a circular data-space into $n = 2^k$ regions consisting of 2^r rings and 2^{k-r} central wedges: $k = 3$, $r = 0, 1, 2, 3$.

rings (leftmost partition in figure 11). These two ways are similar to the concentric hyper-cubes and hyper-pyramids, respectively, which was proposed in [14]. Here, we will explore the possibilities of having partitioning strategies in between these extremes, as illustrated in figure 11, and identify the optimal way of partitioning the circular data-space to minimize the expected number of partitions intersected by a query.

Now we assume that the query radius ρ is small, and calculate the value of r which gives the optimal partitioning into n partitions. We assume that $n = 2^k$ for some integer $k \geq 0$. We also assume that the query circle is completely contained in the unit circle. In other words, its center lies in the circle with origin as center and of radius $1 - \rho$. Thus the boundary of the unit circle itself need not be taken into account in these calculations.

The types of subdivisions of the unit circle into $n = 2^k$ partitions we choose to consider are parameterized by r, and corresponding to r, the unit disk is first divided into 2^r equal-area concentric rings. Then each of these rings is further divided up by 2^{k-r} central wedges of equal angles for a total of 2^k equal area regions. We will now find the value of $r = 0, 1, \ldots, k$ that minimizes the total boundary.

The radius x_1 of the innermost circle of the partitioning into 2^r rings is found from $\pi x_1^2 = \pi/2^r$ as $x_1 = 1/\sqrt{2^r}$. Similarly, the radius of the i-th innermost circle is $x_i = \sqrt{i}/\sqrt{2^r}$ with perimeter $2\pi\sqrt{i}/\sqrt{2^r}$. The sum of the perimeters of the $2^r - 1$ circles is then

$$\frac{2\pi}{\sqrt{2^r}}(1 + \sqrt{2} + \sqrt{3} + \cdots + \sqrt{2^r - 1}).$$

To this sum, we add the lengths of the 2^{k-r} radii that form the boundary of the wedges, to obtain an expression for the total boundary in terms of r as

$$\frac{n}{2^r} + \frac{2\pi}{\sqrt{2^r}}(1 + \sqrt{2} + \sqrt{3} + \cdots + \sqrt{2^r - 1}). \tag{17}$$

To get an idea about the magnitude of r that minimizes this expression, we approximate

$$1 + \sqrt{2} + \sqrt{3} + \cdots + \sqrt{x - 1} \approx \int_0^x \sqrt{t}\, dt = \frac{2}{3}x^{3/2}$$

and minimize the real-valued function of x given by $\frac{n}{x} + \frac{4\pi}{3\sqrt{x}}\, x^{3/2}$ on $1 \le x \le n$. By calculus, we find that the minimum is achieved at $x = \frac{1}{2}\sqrt{\frac{3}{\pi}}\sqrt{n} \approx 0.489\sqrt{n}$. This means that the optimal exponent r for $n = 2^k$ is roughly

$$r \approx \frac{1}{2}k - 1. \tag{18}$$

For small values of k and $n = 2^k$, the optimal values of r that minimizes the expression in (17) can be calculated numerically. The corresponding value of 2^r is the number of rings, and $n/2^r$ is the number of wedges that the unit circle needs to be divided into to minimize the boundary of the n equal-area partitions. The values of the optimal exponent r calculated by brute force from (17) for $k \le 19$ are given in the following table.

.

k	1	2	3	4	5	7	8	9	10	11	12	13	14	15	16	17	18	19
r	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8

These values are in perfect agreement with the formula for r given in (18). In particular when n is of the form $n = 4^k$, then the optimal boundary subdivision has 2^{k-1} rings and 2^{k+1} wedges.

6.3. Implementation of circular partitioning

When we subdivide the unit disk in the plane into 2^k partitions by first dividing into 2^r equi-area rings, and then dividing each ring into 2^{k-r} by means of equi-area central wedges, each partition is determined by 4 parameters. A pair of angles θ_1, θ_2 determines the wedge that the partition is in, and a pair of radii r_1, r_2 determines which ring the partition is in. The angles satisfy $0 \le \theta_1 < \theta_2 \le 2\pi$. The boundary case $\theta_1 = 0$, and $\theta_2 = 2\pi$ is interpreted as the partition in which there are no wedges (i.e. $r = k$ and the partitions consist only of rings). The two radii satisfy $0 \le r_1 < r_1 \le 1$. The extreme cases $r_1 = 0$ and $r_2 = 1$ correspond to the partition in which there are no rings (i.e. $r = 0$ and the partitions consist only of wedges).

The partitions are labeled from 0 to $2^k - 1$ as follows. First of all, label the 2^r rings from 0 to $2^r - 1$ by increasing the radius from the origin to the boundary of the unit circle. In each ring, label the 2^{k-r} pieces determined by the wedges from 0 to $2^{k-r} - 1$ counterclockwise, starting with the wedge that is incident to the horizontal axis in the first quadrant. An integer m with $0 \le m < 2^k$ can be written uniquely in the form $m = q2^r + s$ with $0 \le q < 2^{k-r}$, and $0 \le s < 2^r$. The pair (q, s) uniquely corresponds to the partition in the q-th wedge of the s-th ring of the partitioning under this numbering scheme. For example the partition labeled 7 in figure 12 is encoded as the pair (3, 1), since $7 = 3 \times 2 + 1$. Aside from the extreme

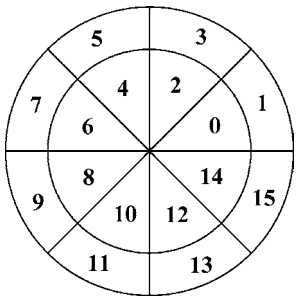


Figure 12. Labeling of the partitions: $n = 2^4$ and $r = 1$.

cases of $r = 0$ and $r = k$, the boundary of the m-th partition is described analytically by the radii

$$r_1 = \frac{\sqrt{s}}{\sqrt{2^r}}, \quad r_2 = \frac{\sqrt{s+1}}{\sqrt{2^r}},$$

and the two angles

$$\theta_1 = \frac{2\pi q}{2^{k-r}}, \quad \theta_2 = \frac{2\pi(q+1)}{2^{k-r}}.$$

6.4. *Experimental results for partitioning*

We have conducted experiments to calculate the expected number of partitions intersected by a randomly selected query circle of radius ρ in the unit circle for varying values of ρ. The experiments for the calculation of the expected values were conducted with three parameters: $n = 2^k$, r, and the radius ρ of the query circle. The data-space is divided into 2^k partitions by first dividing into 2^r equi-area rings, and then dividing each ring into 2^{k-r} by means of equi-area central wedges, $r = 0, 1, \ldots, k$.

In the first set of experiments, we chose the number of partitions to be $n = 2^{12}$, i.e. $k = 12$. When $r = 0$, then the partitioning is just the set of wedges without any concentric rings, i.e., the leftmost partition in figure 11. When $r = 12$, the partitions are formed by circles only, creating 2^{12} concentric rings as partitions, analogous to the rightmost partition in figure 11. In the experiments the query center is chosen randomly in the data-space from the uniform distribution in such a way that the query region lies entirely within the data-space.

We start with query radius $\rho_0 = \frac{1}{\sqrt{n}}$, which gives the query circle the same areas as the area of each partition. Figure 13 illustrates the results of the experiments run with this initial value of ρ. The horizontal axis gives the values of r ranging from 0 to 12, and the vertical axis is the average number of partitions intersected by a random query of radius ρ.

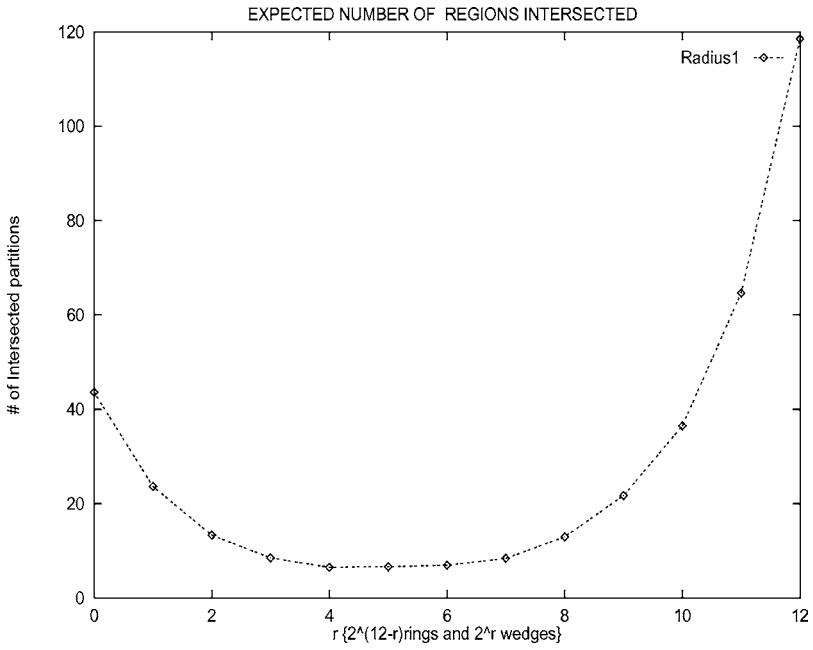


Figure 13. Effect of the number of wedges versus rings on the average number of intersected partitions. Query radius is $\rho_0 = 1/\sqrt{n}$.

This quantity is proportional to the cost of retrieval from secondary storage, since the cost of a query depends on the number of buckets retrieved as a result of the query, and this is exactly the number of partitions which intersect the query circle. When $r = 4$ and $r = 5$ the number of intersected partitions is found to be minimized with expected number $E = 6.55$ and $E = 6.65$, respectively. This result is in agreement with the theoretical analysis. When $r = 12$, the average number of partitions intersected by the queries is $E = 118.55$, about 18 times greater than the average number of partitions intersected when $r = 5$.

We vary the radius of the query. The initial value of $\rho_0 = \frac{1}{\sqrt{n}}$ was reduced by a factor of $\frac{1}{2}$ in each subsequent set of experiments. The corresponding radii are $\rho_1 = \frac{1}{2\sqrt{n}}$, $\rho_2 = \frac{1}{2^2\sqrt{n}}$ etc., until the smallest radius value $\rho_4 = \frac{1}{2^4\sqrt{n}}$. For each radius value ρ, we generate random queries and find the number of partitions intersected by a query circle of radius ρ centered at the query point. In figure 14 the horizontal axis is ρ_i, where $0 \leq i \leq 4$, and the vertical axis is the average number of partitions retrieved by the queries in the case of (1) optimal way of partitioning, (2) partitioning based on our approximation, (3) fully-wedged partitioning (leftmost partition in figure 11), and (4) fully-concentric partitioning (rightmost partition in figure 11). The partitioning technique based on the theoretical analysis gives optimal cost, i.e. optimal number of intersected partitions, for most of the queries. Our theoretical analysis is in perfect agreement with the experiments. The number of partitions retrieved by the queries in concentric and fully-wedged partitioning is much more than the hybrid approach with the appropriate parameters developed in the previous sections.

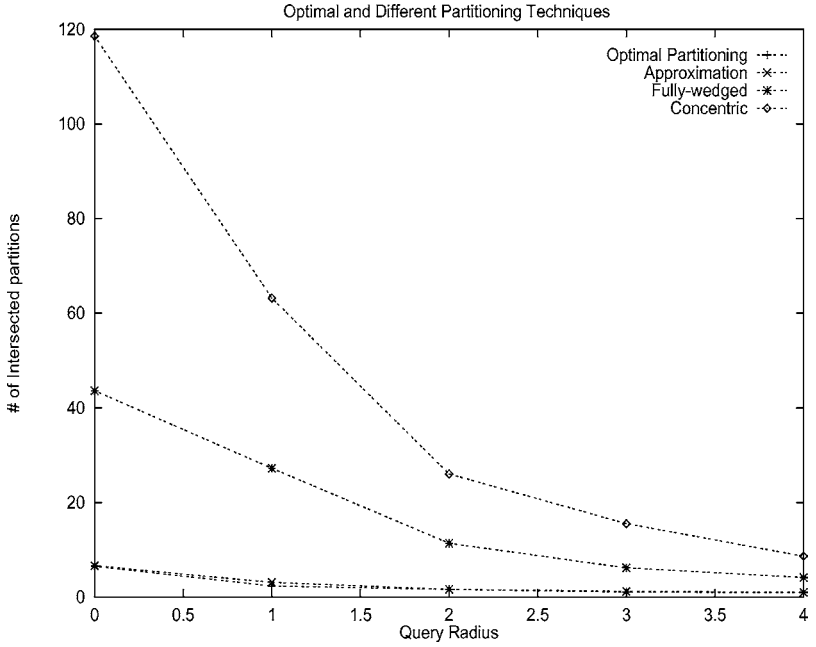


Figure 14. The performance of different partitioning techniques with varying query radius. The point i on the horizontal axis denotes query radius $\rho_i = \frac{1}{2^i\sqrt{n}}$.

6.5. Disk allocation in circular data space partitioning

Cyclic allocation can be applied to circular data space partitioning. We focus only on disk allocation but a page allocation function can be treated in the same way. As in the general case, we assign neighboring buckets to different disks. Direct neighbors are the partitions that have a common boundary, e.g. in figure 12 direct neighbors of partition 2 are 0, 3, and 4. Indirect neighbors are defined as partitions that share a point, e.g. the indirect neighbors of partition 3 are partitions 0 and 4.

Our data-space is partitioned into 2^k equi-area regions determined by 2^r concentric rings and 2^{k-r} central wedges where $r = \lfloor \frac{1}{2}k \rfloor - 1$. As mentioned before, each partition corresponds to an ordered pair of integers (q, s), where s is the rank of the ring ($0 \leq q < 2^r$), and q is the rank of the wedge inside that ring ($0 \leq s < 2^{k-r}$). For example, in figure 12 $(0, 0)$ corresponds to the partition labeled 0, the pair $(2, 1)$ corresponds to the partition labeled 5, and $(5, 0)$ corresponds to the partition 10. In general, the pair (q, s) represents the partition labeled $q2^r + s$. The allocation technique we propose is as follows. Given the number M of available disks, we use a generic allocation technique parameterized by a skip value H. The partition $(0, 0)$ are assigned to device 0. Next, the partitions along this ring (with rank $s = 0$) is assigned to consecutive devices, i.e. partition $(q, 0)$ is assigned to device q mod M. Each partition labeled $(q, 1)$ in ring 1 is assigned to device $(H + q)$ mod M,

which is H devices away from the device on which the partition from the same wedge level in ring 0 was assigned. In general, a partition (q, s) on ring s is assigned to device $(Hq + s) \bmod M$. This assignment is an extension of Cyclic Allocation applied to our partitioning. Cyclic allocation was originally proposed for regular grid partitioning and its performance depends on the H value that is used. The techniques discussed in [26] to determine appropriate H values can also be used here. All other declustering for grid partitioning can be applied here in the same way.

A simple round-robin allocation achieves the optimal I/O cost both for concentric and fully-wedge partitioning. The hybrid partitioning with circles and wedges can be evaluated using a mapping from grid partitioning. The maximum degree of parallelism is achieved when buckets that are retrieved together are spread among all available disks as uniformly as possible. However, since this optimality is not achievable in general, [3] introduced a relaxed optimality definition which ensures that any two buckets that are direct or indirect neighbors of each other are allocated to different disks. Such an allocation is called *near-optimal*. The cyclic allocation is shown to be efficient for the first metric, and optimal for the second one, i.e., it is possible to find appropriate skip values such that no two direct or indirect neighbors are allocated to the same disk, if $2d$ disks are available for d dimensional data. Excluding the border and innermost partitions, e.g. $0, 2, 4, \ldots, 14$, the direct and indirect neighboring bucket pairs remain same in regular grid partitioning and in the partitioning proposed here. Since the cyclic technique guarantees the distribution of the direct and indirect neighbors to different disks in a regular grid partitioning, our allocation technique on the new partitioning scheme also guarantees this. Therefore, the disk allocation described in this section is also near-optimal. Similarly, we expect the same good performance of cyclic allocation in this context. To avoid the neighboring problem for the innermost partitions, we can just add one more partition in the center of the data-space. This additional partition has two advantages. First, we eliminate the case that a single point, i.e. the center, is contained in all 2^{n-r} central wedges. This eliminates the retrieval of all 2^{n-r} innermost partitions for a query point involving the center. Second, the innermost non-adjacent partitions are no longer direct neighbors. Figure 15 illustrates this possible extension.

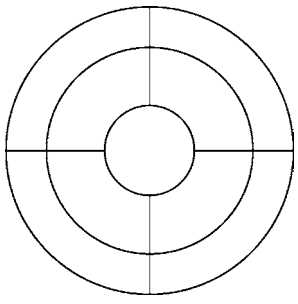


Figure 15. Additional partition in the center.

7. Discussion

This paper explored optimal partitioning techniques for different types of queries on spatial data sets, focusing on partitioning methods that tile the data space without holes and overlaps, and therefore have simple hashing schemes. We discussed a number of instances and computed the expected number of pages retrieved for circular and rectangular queries. One conclusion is that hexagonal partitioning has optimal I/O cost for circular queries over all possible non-overlapping partitioning techniques that use convex regions. This optimality basically comes from the fact that hexagonal partitioning minimizes the total boundary length among all such partitioning strategies. This result was then extended and proved to be valid for small queries over a wider range of partitioning schemes. We also showed that hexagonal partitioning has less I/O cost than the traditional grid for a more general class of square queries where the query is rotated with an angle. It is, however, interesting to note that for the special case of rectilinear square (or rectangular) queries, the traditional grid partitioning provides superior performance. This could be explained by the symmetric relationship between the rectilinear square query and the rectilinear square page. This may also indicate why for traditional relational database applications only rectangular grid partitioning were considered [10, 26]: in a relational database, a select operation specifies a range in each dimension or attribute which corresponds to a rectilinear rectangle. Novel spatial applications need more general query structures. Our results indicate that for such applications, a hexagonal partitioning of the space is appropriate. Hexagonal partitioning is effective for circular and rectangular queries, and it can be used as an alternative to regular grid partitioning with no major changes on the existing algorithms. For instance, a widely used application of regular grid partitioning is declustering where non-overlapping partitions are created and distributed to multiple disks for efficient I/O. It is possible to adapt the algorithms that were developed for regular grid partitioning [10, 22, 26] for hexagonal partitioning [15] and develop techniques for the storage and retrieval of hexagonal partitioning in multi-disk environments. In conclusion, the techniques developed in this paper consider objectives that are crucial for multi-disk searching: minimizing the number of page accesses during the execution of the query, maximizing I/O parallelism, and minimizing disk-arm movement (seek time).

Appendix

A. Total boundary length

In figure 16(a) the data space is the unit square, divided into sixteen partitions by using six line segments each of length 1, three vertical and three horizontal line segments not counting the boundary of the data space. The total length of the boundaries of this partitioning, excluding boundary of the data space is 6, while the total boundary length in figure 16(b) is 8.

In regular non-overlapping partitioning, as shown in figure 8(a), each edge is shared by two partitions, excluding the ones in the border. Therefore, the total boundary used to tile a data space into n partitions is proportional to the number of partitions that are used

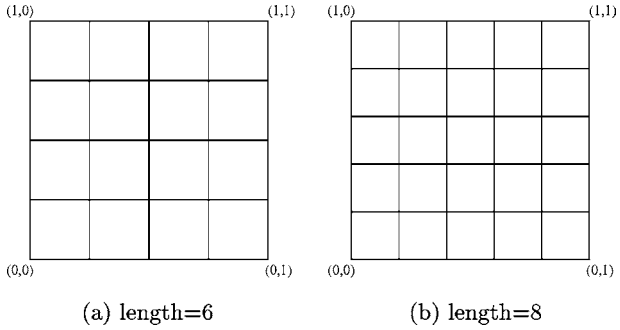


Figure 16. Total boundary length.

and the perimeter of each partition. It has already been observed that the total boundary of the partitions is a factor in determining the average number of page accesses for small circular queries [11]. Our analysis in this paper generalizes this result for all possible sizes of circular queries by showing that the technique that minimizes the total boundary of the partitions minimizes the expected number of partitions intersected by circular queries of any size.

B. Perimeter comparison of the three basic shapes

As we noted, circles can not be used for non-overlapping partitioning of the data space. However, the circle has an *isoperimetric* property known since ancient times: it covers more area compared to its perimeter than any other shape. For a perimeter of 4 units, triangle covers $\frac{2\sqrt{3}}{3}$, square covers 1, and hexagon covers $\frac{4\sqrt{3}}{9}$ units of area. In other words, for a unit area covered by the square, the triangle covers approximately 0.77 units of area and hexagons covers approximately 1.155 units of area.

The honeycomb observation shows the optimality of regular hexagonal partitioning among all equal area partitioning methods. A comparison of hexagonal partitioning with the three basic shapes is given as an example of this observation. We compare the perimeter of a triangle, a square, and a hexagon covering the same area. If the area of these shapes is fixed, the hexagon is the one that has minimum perimeter. For a fixed area of $1/16$ units square, the square has a perimeter of 1, the triangle has a perimeter of 1.14 and the hexagon has a perimeter of 0.93 units (figure 17). Therefore, to cover a fixed area a hexagon has a boundary length of approximately 93% of the one that is used in a square. Therefore, the total boundary length of n hexagonal partitions that are used to cover a given area is again approximately 93% of the total boundary length of n square partitions. Even though the area of each partition and the total area to be covered in each partitioning is same, the total boundary length is minimized with hexagonal partitioning.

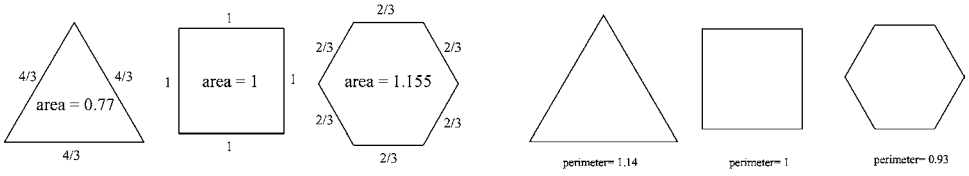


Figure 17. Area and perimeter comparison of three basic shapes.

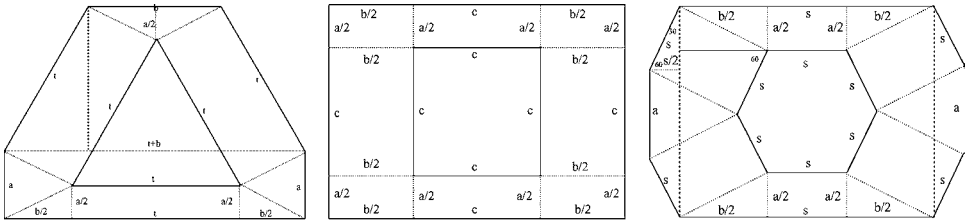


Figure 18. Minkowski sums w.r.t. rectangles.

C. Rectilinear rectangular range queries

In this section, we compare the triangular, square, and hexagonal grid partitioning with respect to the expected number of page intersections for rectilinear rectangular queries with side lengths (a, b). Figure 18 shows the Minkowski sum for each shape with equal area with respect to the rectilinear rectangular query (a, b), i.e., $M(triangle_t, rectangle_{a,b})$, $M(square_c, rectangle_{a,b})$, and $M(hexagon_s, rectangle_{a,b})$.

As can be seen from the figures, we have $M(square_c, square_a) = (c + a)(c + b)$. Substituting $c = s\sqrt{\frac{3\sqrt{3}}{2}}$ we find the Minkowski Sum of the square page as follows:

$$M(square_c, rectangle_{a,b}) = \frac{3\sqrt{3}}{2}s^2 + (a + b)\sqrt{\frac{3\sqrt{3}}{2}}s + ab \tag{19}$$

The total area of the $M(hexagon_s, square_a)$ is the area of the rectangle inside the Minkowski sum M plus the two trapezoids created on both sides of M. Therefore,

$$M(hexagon_s, rect_{a,b}) = (s + b)(s\sqrt{3} + a) + \frac{(2a + s\sqrt{3})s}{2} \tag{20}$$

$$M(hexagon_s, rect_{a,b}) = \frac{3\sqrt{3}}{2}s^2 + 2as + bs\sqrt{3} + ab \tag{21}$$

Similarly,

$$M(triangle_t, rect_{a,b}) = \frac{3\sqrt{3}}{2}s^2 + \sqrt{6}as + \frac{3\sqrt{2}}{2}bs + ab \tag{22}$$

Comparing (21) and (22), we have $M(hexagon_s, rect_{a,b}) < M(triangle_t, rect_{a,b})$. To compare Eqs. (19) and (21), we need to compare the different terms, i.e., $(a+b)\sqrt{\frac{3\sqrt{3}}{2}}s$ and $2as + bs\sqrt{3}$. Substituting $\sqrt{\frac{3\sqrt{3}}{2}} \approx 1.612$ in both equations, we find that $M(square_c, rectangle_{a,b}) < M(hexagon_s, rectangle_{a,b})$. Similar to the previous analysis, we conclude that

$$E_{\text{square}}(rectangle_{a,b}) < E_{\text{hexagon}}(rectangle_{a,b}) < E_{\text{triangle}}(rectangle_{a,b}).$$

For rectilinear rectangular range queries square grid partitioning outperforms hexagonal and triangular partitioning in terms of the expected number of page accesses.

Acknowledgments

Hakan Ferhatosmanoglu is supported in part by the U.S. Department of Energy (DOE) Award No. DE-FG02-03ER25573. Divyakant Agrawal and Amr El Abbadi are supported in part by NSF Award No. IIS 02-09112 and IIS 02-23022. However, any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect the views of the sponsors.

References

1. M.J. Atallah and S. Prabhakar, "(Almost) optimal parallel block access for range queries," in Proc. ACM Symp. on Principles of Database Systems, Dallas, Texas, May 2000, pp. 205–215.
2. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R∗ tree: An efficient and robust access method for points and rectangles," in Proc. ACM SIGMOD Int. Conf. on Management of Data, May 23–25, 1990, pp. 322–331.
3. S. Berchtold, C. Bohm, B. Braunmuller, D.A. Keim, and H-P. Kriegel, "Fast parallel similarity search in multimedia databases," in Proc. ACM SIGMOD Int. Conf. on Management of Data, Arizona, USA, 1997, pp. 1–12.
4. S. Berchtold, C. Bohm, D. Keim, and H.-P. Kriegel, "A cost model for nearest neighbor search," in Proc. ACM Symp. on Principles of Database Systems, Tuscon, Arizona, USA, June 1997, pp. 78–86.
5. S. Berchtold, C. Bohm, and H.-P. Kriegel, "The Pyramid-Technique: Towards breaking the curse of dimensionality," in Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, Washington, USA, June 1998, pp. 142–153.
6. R. Bhatia, R.K. Sinha, and C.-M. Chen, "Hierarchical declustering schemes for range queries," in Advances in Database Technology—EDBT 2000, 7th International Conference on Extending Database Technology, Lecture Notes in Computer Science, Konstanz, Germany, March 2000, pp. 525–537.
7. C.-M. Chen, R. Bhatia, and R. Sinha, "Declustering using golden ratio sequences," in International Conference on Data Engineering, San Diego, California, Feb 2000, pp. 271–280.
8. C.-M. Chen and C.T. Cheng, "From discrepancy to declustering: Near optimal multidimensional declustering strategies for range queries," in Proc. ACM Symp. on Principles of Database Systems, Wisconsin, Madison, 2002, pp. 29–38.

9. X. Cheng, R. Dolin, M. Neary, S. Prabhakar, K. Ravikanth, D. Wu, D. Agrawal, A. El Abbadi, M. Freeston, A. Singh, T. Smith, and J. Su, "Scalable access within the context of digital libraries," in IEEE Proceedings of the International Conference on Advances in Digital Libraries, ADL, Washington, D.C., 1997, pp. 70–81.
10. H.C. Du and J.S. Sobolewski, "Disk allocation for cartesian product files on multiple-disk systems," ACM Transactions of Database Systems, vol. 7, no. 1, pp. 82–101, 1982.
11. O. Egecioglu and H. Ferhatosmanoglu, "Circular data-space partitioning for similarity queries and parallel disk allocation," in Proc. of IASTED International Conference on Parallel and Distributed Computing and Systems, Nov. 1999, pp. 194–200.
12. C. Faloutsos and P. Bhagwat, "Declustering using fractals," in Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems, San Diego, CA, Jan. 1993, pp. 18–25.
13. H. Ferhatosmanoglu, D. Agrawal, and A. El Abbadi, "Clustering declustered data for efficient retrieval," in Proc. Conf. on Information and Knowledge Management, Kansas City, Missouri, Nov. 1999, pp. 343–350.
14. H. Ferhatosmanoglu, D. Agrawal, and A. El Abbadi, "Concentric hyperspaces and disk allocation for fast parallel range searching," in Proc. Int. Conf. Data Engineering, Sydney, Australia, March 1999, pp. 608–615.
15. H. Ferhatosmanoglu, D. Agrawal, and A. El Abbadi, "Optimal partitioning for efficient I/O in spatial databases," in Proc. of the European Conference on Parallel Computing (Euro-Par), Manchester, UK, Aug. 2001.
16. H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi, "Vector approximation based indexing for non-uniform high dimensional data sets," in Proceedings of the 9th ACM Int. Conf. on Information and Knowledge Management, McLean, Virginia, Nov. 2000, pp. 202–209.
17. V. Gaede and O. Gunther, "Multidimensional access methods," ACM Computing Surveys, vol. 30, pp. 170–231, 1998.
18. A. Guttman, "R-trees: A dynamic index structure for spatial searching," in Proc. ACM SIGMOD Int. Conf. on Management of Data, 1984, pp. 47–57.
19. T.C. Hales, "The honeycomb conjecture. Available at http://xxx.lanl.gov/abs/math.MG/9906042, June 1999.
20. T.C. Hales, "Historical background on hexagonal honeycomb. http://www.math.lsa.umich.edu/ hales/ countdown/honey/hexagonHistory.html, March 2000.
21. J. Hellerstein, E. Koutsoupias, and C. Papadimitriou, "On the analysis of indexing schemes," in Proc. ACM Symp. on Principles of Database Systems, Tucson, Arizona, June 1997, pp. 249–256.
22. K.A. Hua and H.C. Young, "A general multidimensional data allocation method for multicomputer database systems," in Database and Expert System Applications, Toulouse, France, Sept. 1997, pp. 401–409.
23. M.H. Kim and S. Pramanik, "Optimal file distribution for partial match retrieval," in Proc. ACM SIGMOD Int. Conf. on Management of Data, Chicago, 1988, pp. 173–182.
24. J. Nievergelt, H. Hinterberger, and K.C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," ACM Transactions on Database Systems vol. 9, no. 1, pp. 38–71, 1984.
25. A. Okabe, B. Boots, K. Sugihara, and S. Nok Chiu, Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, Wiley, 2001.
26. S. Prabhakar, K. Abdel-Ghaffar, D. Agrawal, and A. El Abbadi, "Cyclic allocation of two-dimensional data," in International Conference on Data Engineering, Orlando, Florida, Feb. 1998, pp. 94–101.
27. S. Prabhakar, D. Agrawal, and A. El Abbadi, "Efficient disk allocation for fast similarity searching," in 10th International Symposium on Parallel Algorithms and Architectures, SPAA'98, Puerto Vallarta, Mexico, June 1998, pp. 78–87.
28. J.T. Robinson, "The kdb-tree: A search structure for large multi-dimensional dynamic indexes," in Proc. ACM SIGMOD Int. Conf. on Management of Data, 1981, pp. 10–18.
29. H. Samet, The Design and Analysis of Spatial Structures. Addison Wesley Publishing Company, Inc., Massachusetts, 1989.
30. A.S. Tosun and H. Ferhatosmanoglu, "Optimal parallel I/O using replication," in Proceedings of International Workshops on Parallel Processing (ICPP), Vancouver, Canada, Aug. 2002.
31. R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in Proceedings of the Int. Conf. on Very Large Data Bases, New York City, New York, Aug. 1998, pp. 194–205.
32. D. White and R. Jain, "Similarity indexing with the SS-tree," in Proc. Int. Conf. Data Engineering, 1996, pp. 516–523.