Optimal Partitioning for Spatial Data

Hakan Ferhatosmanoglu, Divyakant Agrawal, Amr El Abbadi
Department of Computer Science, University of California
Santa Barbara, CA 93106
{hakan,agrawal,amr}@cs.ucsb.edu

Abstract

It is desirable to design partitioning techniques that minimize the I/O time incurred during query execution in spatial databases. In this paper, we explore optimal partitioning techniques for spatial data for different types of queries. In particular, we show that hexagonal partitioning has optimal I/O cost for circular queries compared to all possible non-overlapping partitioning techniques that use convex regions. For rectangular queries, we show that although for the special case when queries are rectilinear, rectangular grid partitioning gives superior performance, hexagonal partitioning has overall better I/O cost for a general class of range queries. We also discuss storage and retrieval techniques for hexagonal partitioning using current techniques for rectangular grid partitioning.

1 Introduction

Efficient retrieval of multidimensional data has gained importance by the recent developments in information technology. Spatial databases and Geographical Information Systems (GIS) [5] are typical applications of multidimensional data. In these applications, the data objects are represented as two-dimensional feature vectors, and the similarity between objects are defined by a distance function between corresponding feature vectors. Several index structures have been proposed for retrieval of spatial data [20, 11, 16, 19, 12, 1]. The common approach is to group the objects according to their spatial locations and store the created groups as pages in physical storage.

Most of the approaches in the literature for indexing spatial data are based on clustering data points with a rectangular organization [20, 11, 12, 1]. There have been several approaches for the storage and retrieval of spatial data based on partitioning the data space. Grid based file structures have been very effectively used to index spatial data [16], and there have been several approaches based on the grid partitioning [20, 11]. Because of their simplicity in hashing and mapping to physical storage, *regular* equi-sized partitioning are widely used for retrieval and storage of spatial data. A common example of such techniques is the regular grid partitioning. One very important application of regular grid partitioning is multi-disk *declustering* of spatial data. First, the data space is partitioned into disjoint regular rectangular partitions. Then the data partitions or buckets are allocated to multiple I/O devices such that neighboring partitions are allocated to different disks. Performance improvements for queries occur when the buckets involved in query processing are stored on different disks, and hence can be retrieved in parallel. Numerous methods have been proposed: Disk Modulo (DM) [6], Hilbert (HCAM) [8], Near Optimal Declustering (NoD) [2], General Multidimensional Data Allocation (GMDA) [15] and the Cyclic Allocation Schemes [17]. All of these approaches were based on *non-overlapping* rectangular partitions. Another successful application of regular partitioning is the vector approximation (VA) based indexing for multi-dimensional data [21, 10]. The VA-file approach divides the data space into 2^b rectangular partitions where b is the total number of bits specified by the user. Each cell has a bit representation of length b which approximates the data points that fall into a cell by the corresponding bit representation of the cell.

Rectangles are the most common geometrical shape that have been used to index spatial data. Besides rectangles, there have been a number of proposals for using different shapes for indexing. Recently circles were proposed for multi-dimensional indexing especially for similarity searching [22]. One major problem with circles is that it is not possible to partition the data space with circles without overlaps and holes. These overlaps cause severe degradation in the query performance. If all data points are covered by reducing the holes, an increasing number of overlapping circular partitions is needed. Large numbers of partitions and overlaps between them cause a large number of intersections and this directly increases the I/O cost while querying. In several applications, it is desirable to design partitioning techniques that tile the data space without holes and overlaps, and therefore have simple hashing schemes. Therefore rectangles have become more effective than circles in many applications especially for range queries on spatial data.

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. In a range query, the pages that may have the query result need to be retrieved from secondary storage. As we will establish, 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 [9, 4]. It is therefore very important to develop partitioning techniques which minimize the expected number of partitions retrieved by a query and hence the number of disk accesses. It is well known that rectangles are effective for non-overlapping partitioning. An important question to ask 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 techniques for different types of queries. In particular, we show that hexagonal partitioning has optimal I/O cost for circular queries compared to all possible non-overlapping partitioning techniques that use convex regions. For rectangular queries, we show that although for the special case when queries are rectilinear 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 techniques for storage and retrieval of hexagonal partitioning by applying the techniques developed for rectangular grid partitioning.

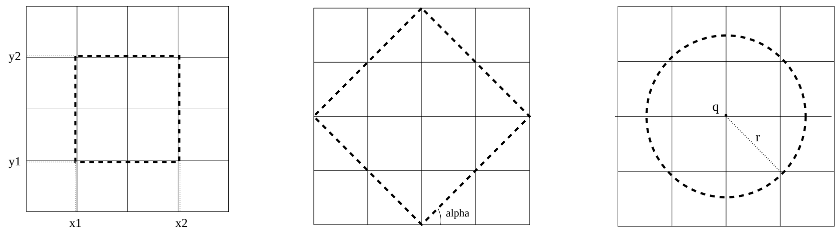
In Section 2 we discuss the importance of data organization on minimizing the I/O cost of spatial queries and summarize alternative partitioning techniques. In Section 3, we analyze the partitioning techniques with respect to the I/O cost of spatial queries and show the optimal partitioning technique for each query type. Section 4 discusses techniques for storage and retrieval of hexagonal partitioning using current techniques for rectangular grid. Section 5 includes the final discussion and conclusions.

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, the query point is specified and all points "similar" to the query point are retrieved. A popular related query is k-nearest neighbor query. The k-nearest neighbor, $k-NN$, problem is defined as finding the k nearest points to the query point q. Traditional range queries have been specified using rectilinear queries, i.e., rectangular with sides parallel to 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 range in each dimension. 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)). A commonly used query is the circular range query which is also called as ϵ-similarity query. It specifies a query point q and and a radius r, $Q_c = (q, r)$, which defines 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 different types of similarity queries using different metrics. For example, the circular range query corresponds to similarity in L_2 metric, the rectilinear rectangular range query corresponds to similarity in the L_∞ metric and the diamond range query corresponds to similarity in the L_1 metric.

In both range and similarity queries, the pages that have the possibility to contain a portion of the query result are retrieved as a result of the query. As mentioned before, the spatial data objects are grouped according to their spatial locations and they are stored as pages in physical storage. Each page represents a spatial location in the data space. Therefore, in range queries the pages that are intersected by a query are retrieved as a result of the query. Similarly, $k-NN$ queries have to retrieve all pages that intersect the circle with the query point as the center and the distance between the query point and the $k-NN$ point as the radius. Even if there is a small portion of intersection of a page with the query



(a) Rectilinear Query

(b) Diamond Query

(c) Circular Query

Figure 1: Range queries intersecting all partitions

region, the page needs to be retrieved since it may contain relevant data. Is it possible to reduce this I/O cost by changing the initial page organization of the data space? What is the best possible that one can do? In general, assuming the data space has an area of 1, the optimal cost for a range query with area A is equal to $\lceil A.p \rceil$, where p is the total number of pages. Although, it is not possible to achieve minimal cost for every range query, a careful investigation is needed to reduce the I/O cost as much as possible. The number of pages accessed plays an important role in the response time incurred by range queries. The key point to minimize the I/O cost for queries depends on how data points are organized into 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. In general, an organization of a database is *I/O optimal* if and only if it minimizes the expected number of page accesses incurred by the queries.

Regular equi-sized partitioning are widely used for organization of spatial data. They provide very efficient functions for hashing and mapping the partitions to pages in storage. The general approach is first to tile the data space into disjoint regular rectangular partitions and then to map each partition to a physical page in the storage. Are there any alternative techniques that tile the data space with non-overlapping partitions of identical shapes? It is known that it is not possible to tile the space with regular non-overlapping identical convex polygons with edges of more than 6 and there are only three basic tile shapes (triangles, rectangles, and hexagons) for regular partitioning (see Appendix A for the proof). We will consider these three basic shapes and compare their behavior under various conditions. Equilateral triangle, square, and regular hexagon are considered for triangle, rectangle, and hexagon respectively. All the techniques discussed in this paper have simple hashing properties that can be used to map the partitions to physical storage easily. We note that, although we consider these possible partitionings for our analysis, the optimality results in this paper are not restricted to these shapes.

3 I/O Cost for Various Query Types

Given a query, we explore different partitioning techniques that minimize the expected number of pages retrieved. The partitions that have to be accessed are the ones that intersect the range query region. Therefore, we will compute the expected number of partitions that intersect a range query region by using the methodology of Minkowski sum, $MSum$ [3, 21]. In general, the Minkowski sum of two closed regions can be considered as expanding one region by the other. To compute the expected number of partitions that intersect a query, first the region of a partition, which also corresponds to a page region, is enlarged by considering the shape of the query.

Suppose the partitions, which also correspond to the pages, are square shapes of side length c and the queries are circular. The Minkowski sum methodology is used to compute the expected number of partitions that intersect the circular query region as follows. The Minkowski sum of a square page of side length c, with respect to a circular region with a radius of r, is the enlarged object $MSum(square_c, circle_r)$. The enlarged object is created by moving the center of the circular query over the surface of the square (Figure 2(b)). Therefore, it consists of all points that are either in the square page or have a distance to the surface of the page less than r. 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 that the corresponding page is being accessed, $P(page)$. Considering the unit data space, i.e., $[0, 1]^2$, or simply normalizing the areas with respect to the total area, the area of the enlarged object gives the probability of the page to be accessed. Note that, the regions that are created by the Minkowski sum of the partitions in the border of the two-dimensional data space are negligible when compared to the total amount of regions created by Minkowski sums of all partitions. The expected number of pages which are intersected by a query q is $E(q)$ and it is the sum of the probabilities of the pages intersected by the query. If we have regular equi-sized partitioning, i.e., p partitions of identical shape, then 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

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

where p is the number of partitions. For example, for a square partitioning this equation is simply $E_{square}(q) = p \cdot MSum(square_c, q)$.

In this section, we analyze the I/O cost performance of the partitioning techniques for range queries. We will investigate the optimal partitioning that must be used for different conditions to minimize the I/O cost.

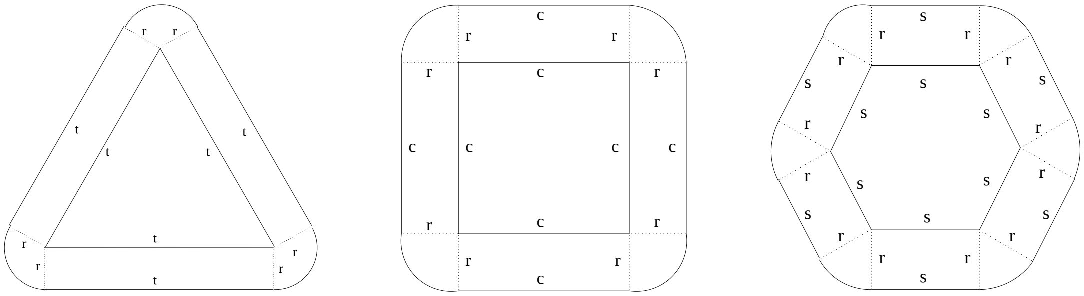
3.1 I/O Cost for Circular Range Queries

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 triangular, square, and hexagonal pages, i.e., $MSum(triangle_t, circle_r)$, $MSum(square_c, circle_r)$, and $MSum(hexagon_s, circle_r)$, 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., $\frac{t^2\sqrt{3}}{4} = c^2 = s^2\frac{3\sqrt{3}}{2} = 1/p$. As can be seen from the figures, we have the following areas for each of Minkowski sums:

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

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

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



(a) Triangle-Circle (b) Square-Circle (c) Hexagon-Circle

Figure 2: Minkowski Sums 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 $\sqrt{6} \approx 2.45$, Equation 2 and Equation 3 become as the following:

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

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

Comparing Equations 4, 5, and 6, we find $MSum(hexagon_s, circle_r) < MSum(square_c, circle_r) < MSum(triangle_t, circle_r)$. This result 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 Equation 1, we finally find $E_{hexagon}(circle_r) < E_{square}(circle_r) < E_{triangle}(circle_r)$. The expected number of partitions intersected by a circular query in a hexagonal grid is always less than the one in a square grid and a triangular grid. This result simply tells us that 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 the page accesses, i.e., I/O operations, occurred during query.

Indeed, we will now generalize the superiority of hexagonal partitioning over other possible approaches and prove that hexagonal partitioning is optimal, among all non-overlapping partitioning techniques of equal area convex regions, with respect to the number 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:*

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

Proof

For a convex polygon of n sides, the $MSum$ region with respect to 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 $180 - \Theta_i$ degrees, where Θ_i is the corresponding angle in the polygon. Since the summation of all these n angles of pies is 360 degrees, $\sum_1^n (180 - \Theta_i) = 180n - 180(n-2) = 360$, 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

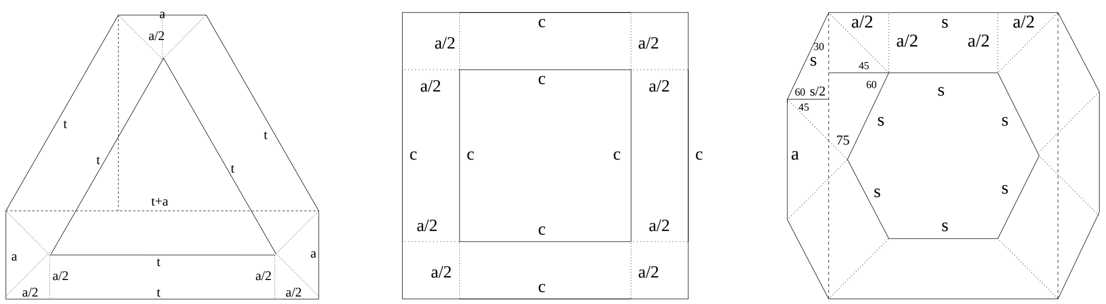
In Equation 7, since r does not change with partitioning and $area(polygon) = 1/p$, the only value that makes a difference is the perimeter M that is used in the pages. This is true for all possible r values that a circular query can take.

□

Minimizing the perimeter of each partition minimizes the total boundary length of the partitioning. The total boundary of the partitions is defined as the length of the boundaries that are used to divide the data-space into partitions (see Appendix B 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 C). What is the optimal way of minimizing the perimeter and the total boundary? Recently, it has been proved that any partitioning of the plane into regions of equal area has perimeter at least that of the regular hexagonal honeycomb tiling [13]. This has known as the classical honeycomb observation and has been the motivation for numerous interesting applications. 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 the nice 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" [14].

By Lemma 1 and 2 we conclude that a 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 compared to all possible equal area partitioning techniques, it also minimizes the expected number of partitions retrieved as a result of a circular query. Hence, we have the following theorem.

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



(a) Triangle-Rectilinear Square

(b) Square-Rectilinear Square

(c) Hexagon-Rectilinear Square

Figure 3: Minkowski Sums w.r.t. Rectilinear Squares

3.2 I/O Cost for Square Range Queries

In this section, we analyze 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 square grid for 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 three shapes and a rectilinear square region (See Appendix D for a similar analysis of rectilinear rectangular range queries). Figure 3 illustrates the Minkowski sum regions for each case. As can be seen from the figures, we have $MSum(square_c, rectilinear_a) = (c+a)^2$. Substituting $c = s\sqrt{\frac{3\sqrt{3}}{2}}$, we find the Minkowski Sum of the square page as follows:

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

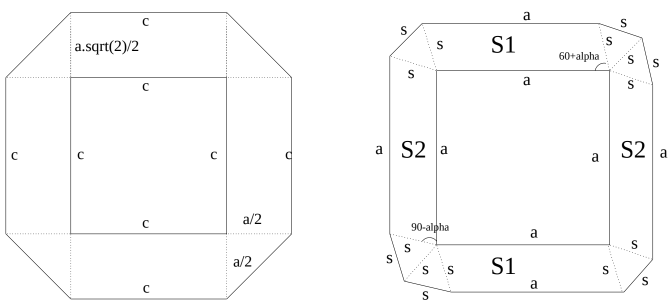
The total area of the $Msum(hexagon_s, restilinear_a)$ is the area of the rectangle inside the $MSum$ plus the two trapezoid created on both sides of the $MSum$. Therefore,

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

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

Similarly,

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



(a) Square-Square

(b) Square-Hexagon (= Hexagon-Square)

Figure 4: Minkowski Sums w.r.t. Squares (with any orientation α)

Comparing the Equations 8, 10, and 11, we have $MSum(square_c, rectilinear_a) < MSum(hexagon_s, rectilinear_a) < MSum(triangle_t, rectilinear_a)$. Therefore, $E_{square}(rectilinear_a) < E_{hexagon}(rectilinear_a) < E_{triangle}(rectilinear_a)$. 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 *general* square queries with any *orientation*. We will compare the hexagonal and square partitioning by starting to compute the Minkowski Sums for square and hexagon tiles and general square region. The square region has an angle α with the $x - axis$. The examples of special cases include the diamond query, i.e., $\alpha = \Pi/4$, and the rectilinear square query, i.e., $\alpha = 0$. In this section, for simplicity in $MSum$, 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 $MSum$, i.e., $MSum(hexagon_s, square_a) = MSum(square_a, hexagon_s)$, for simplicity we illustrate $MSum(square_a, hexagon_s)$ in Figure 4.

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

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

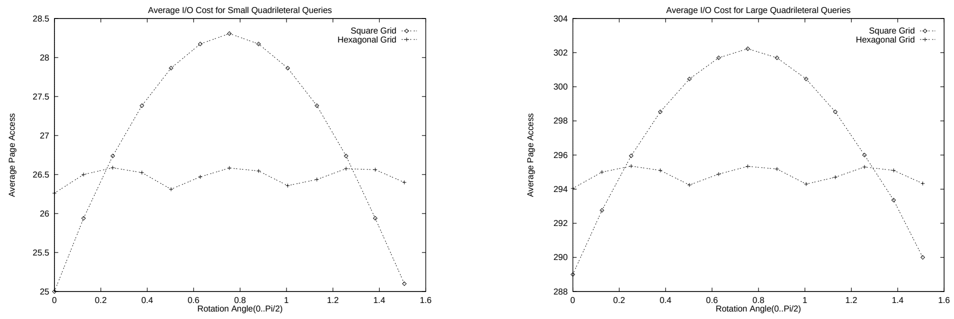
where $0 \leq \alpha \leq \frac{\Pi}{4}$.

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

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

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 the



(a) Small General Square Queries

(b) Large General Square Queries

Figure 5: Average I/O Cost for General Square Queries with varying angles

other angles, because rotating an angle of β not in this range gives the same result as rotating with 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 (Figure 5(a) illustrates results for small size range queries and Figure 5(b) illustrates results for large size range queries). The $x-$axis of the figures corresponds to the angle α in terms of radians. For example, an angle of $\Pi/4$ (≈ 0.785) corresponds the diamond query which is the shape for similarity queries in the L_1 metric. For most of the angles which correspond to different query types, including the diamond query, hexagonal partitioning achieves better performance results by requiring less number of I/O accesses. Since we have the general formula for the square query specified with a center and an angle, we can compute the expected number of partitions intersected by such queries for both square and hexagonal grid. By integrating all such possible queries and computing the expected number by taking the uniform average, we can compute the expected $MSum$, E_{MSum}, of each technique. As shown before, the comparison of $MSums$ will give the comparison of techniques with respect to the expected number of partitions intersected by a random square query. Integrating Equations 12 and 13 for all possible values of α, we compute the expected $MSum$ for each partition.

$$E_{MSum_{square}}(square_a) = \frac{4}{\Pi} \int_{\Pi/4}^{\Pi/2} (MSum(square_1, square_a) = a^2 + 1 + \frac{8\sqrt{2}a}{\Pi} \int_{\Pi/4}^{\Pi/2} sin\alpha d\alpha$$

Therefore,

$$E_{MSum_{square}}(square_a) = 1 + a^2 + \frac{8a}{\Pi} \tag{14}$$

Similarly, for hexagon,

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

Solving this equation, we finally have,

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

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

Comparing Equations 14 and 15, we find $E_{MSum_{hexagon}}(square_a) < E_{MSum_{square}}(square_a)$. Hence, we conclude that the 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 the hexagonal partitioning is effective for range queries, and hence can be used as an effective alternative for regular grid partitioning. In this section, we develop simple hashing, clustering and declustering techniques for hexagonal partitioning adapting the techniques developed for rectangular grid partitioning.

4.1 Hashing

In this subsection, we define a hashing function for regular hexagonal partitioning. The hashing function finds the corresponding hexagonal partitioning of a given data point. Hashing functions for rectangular partitioning are very simple and well-known. We will use rectangular hashing in the development of 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 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 cell is defined by $G(i, j)$.

Obviously, the number of rectangular cells is more than the number of hexagonal cells. Depending on the location, some of the cells 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 cell can be mapped to one or two hexagonal partitions. Therefore, we can hash the data points using hashing on regular grid and find the corresponding hexagon(s). If there are two hexagons an additional simple check is needed to identify the hexagon where the point is located. The different cases for mapping of rectangular grid cells to hexagonal partitions are shown in Figure 7.

4.2 Declustering and Clustering of Hexagonal 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 techniques that are optimized for such environments. In this section, we develop a multi-disk organization of the data using hexagonal partitioning. We develop *declustering* (distributing the neighboring partitions across multiple disks) and *clustering* (storing spatially close partitions close in disk) for hexagonal partitions.

An observation important for declustering is that for two-dimensional regular grids, 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,

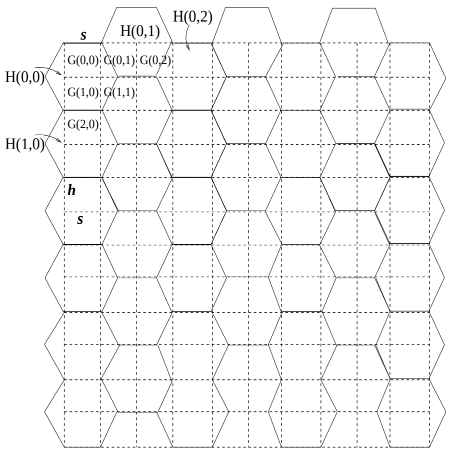


Figure 6: Hexagonal hashing using rectangular grid

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

Figure 7: Mapping of rectangular grid cells to hexagonal partitions

called *counts*, for the quality of neighbor declustering is defined and used in [18, 2]. It is a measure of how far the allocation is from being optimal in terms declustering 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 specifies the disks as well as the page assignments of each partitions on that disk, assuming that there are five disks in the system. The disk assignment in Figure 8(a) is developed by applying the Disk Modulo (DM) declustering [6] for hexagonal partitioning. We assign the pages in the row major order for each disk. In general, given a partition, $PartitionId = (i, j)$, where i is the row number and j is the column number, the physical location of the partition, i.e., the corresponding disk and the page within that disk, can be computed as follows:

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

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 [15, 17]. As an example, Figure 8(b) illustrates the Cyclic Allocation Technique [17] applied on 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 $j \ mod \ M$. Each partition in row 1, $(1,j)$ is assigned to device $(H+j) \ mod \ M$, 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 $(Hi+j)$ $mod \ M$. 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. 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 number of 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 [18]. In a hexagonal partitioning, the neighbors of a bucket (i,j) are the buckets $\{(i-1,j),(i,j-1),(i,j+1),(i+1,j),(i+1,j-1),(i+1,j+1)\}$.

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 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.

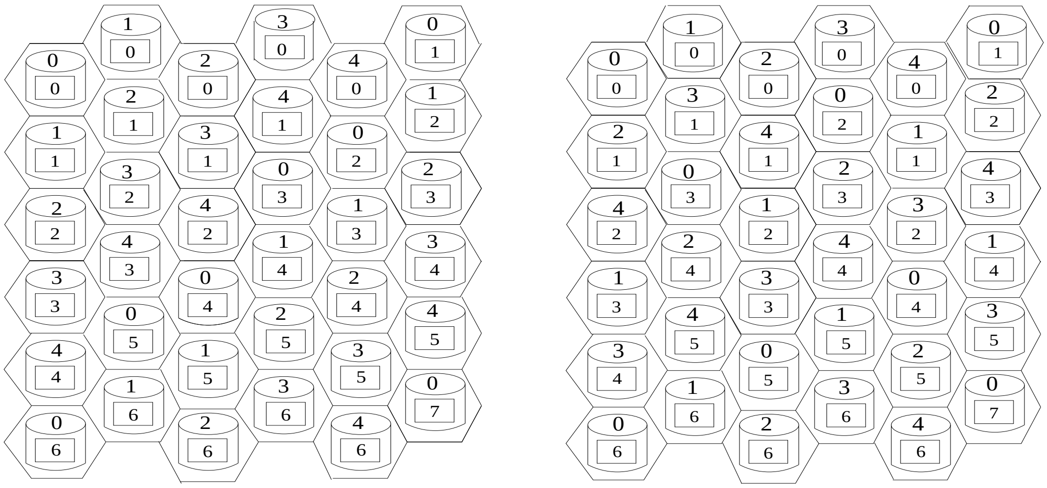
We define the general allocation formula for two dimensional Cyclic Allocation Techniques by the following functions.

$$disk(i,j) = d = (Hi + j) \bmod M$$
$$page(i,j) = \lceil \tfrac{j+1-((d-i.H) \bmod M)}{M} \rceil + \textstyle\sum_{k=0}^{i-1} \lceil \tfrac{N-((d-k.H) \bmod M)}{M} \rceil - 1.$$

The page allocation function is developed based on the disk allocation function. The summation in the page allocation formula computes the number of partitions assigned to disk d for each row starting from the 0^{th} row to $(i-1)^{st}$ row. The first expression computes the same value for the i^{th} row but not considering the partitions that come after partition (i,j). Then we subtract 1, because the page numbering starts with a zero as initial page number. As an example, under DM declustering (which is a Cyclic Scheme with a skip value H of 1), the disk number of $(3,2)$ is found to be 0 and the page number inside the disk is found to be 4 which is also shown in Figure 8(a).

5 Discussion

In this paper, we explored optimal partitioning techniques for different types of queries. We focused on partitioning techniques that tile the data space without holes and overlaps, and therefore have simple hashing schemes. We discussed the possible cases and computed the expected number of pages retrieved for circular and rectangular queries. We showed that hexagonal partitioning has optimal I/O cost for circular queries over all possible non-overlapping partitioning techniques that use convex regions. We also show that hexagonal partitioning has better I/O cost than the traditional grid for the general class of square queries. 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 due to 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 [6, 17]. 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 with queries of any orientations. Our results indicate that for such applications, a hexagonal partitioning of the space should be used. The hexagonal partitioning is shown to be effective for circular and rectangular queries. It can be used as an effective alternative for 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. We showed how to adapt the techniques that were developed for regular grid partitioning [6, 15, 17] for hexagonal partitioning.



(a) DM declustering on hexagons

(b) Cyclic declustering on hexagons

Figure 8: Declustering of hexagonal partitions

References

[1] 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*, pages 322–331, May 23-25 1990.

[2] 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*, pages 1–12, Arizona, U.S.A., 1997.

[3] 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*, pages 78–86, Tuscon, Arizona, USA, June 1997.

[4] 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*, pages 142–153, Seattle, Washington, USA, June 1998.

[5] 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*, pages 70–81, Washington, D.C., 1997.

[6] H. C. Du and J. S. Sobolewski. Disk allocation for cartesian product files on multiple-disk systems. *ACM Transactions of Database Systems*, 7(1):82–101, March 1982.

[7] 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*, pages 194–200, November 1999.

[8] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18 – 25, San Diego, CA, Jan 1993.

[9] H. Ferhatosmanoglu, D. Agrawal, and A. El Abbadi. Concentric hyperspaces and disk allocation for fast parallel range searching. In *Proc. Int. Conf. Data Engineering*, pages 608–615, Sydney, Australia, March 1999.

[10] 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*, pages 202–209, McLean, Virginia, November 2000.

[11] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, 1998.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[13] Thomas C. Hales. The honeycomb conjecture. available at http://xxx.lanl.gov/abs/math.MG/9906042, June 1999.

[14] Thomas C. Hales. Historical background on hexagonal honeycomb. http://www.math.lsa.umich.edu/ hales/countdown/honey/hexagonHistory.html, March 2000.

[15] K. A. Hua and H. C. Young. A general multidimensional data allocation method for multicomputer database systems. In *Database and Expert System Applications*, pages 401–409, Toulouse, France, September 1997.

[16] J. Nievergelt, H. Hinterberger, and K.C.Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems 9*, 1:38–71, March 1984.

[17] S. Prabhakar, K. Abdel-Ghaffar, D. Agrawal, and A. El Abbadi. Cyclic allocation of two-dimensional data. In *International Conference on Data Engineering*, pages 94–101, Orlando, Florida, Feb 1998.

[18] 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*, pages 78–87, Puerto Vallarta, Mexico, June 1998.

[19] 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*, pages 10–18, 1981.

[20] H. Samet. *The Design and Analysis of Spatial Structures.* Addison Wesley Publishing Company, Inc., Massachusetts, 1989.

[21] 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*, pages 194–205, New York City, New York, August 1998.

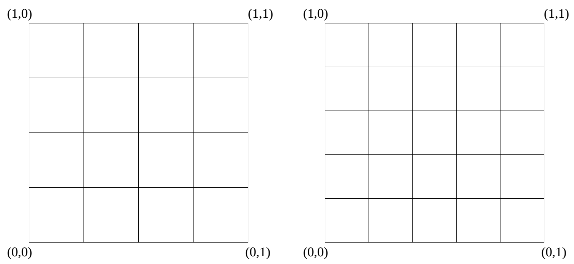
[22] D. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. Int. Conf. Data Engineering*, pages 516–523, 1996.

A Three Basic Shapes for Regular Partitioning

It is not possible to tile the space with regular non-overlapping identical convex polygons with edges of more than 6. This result can easily be proved by considering the angle at each intersection corner of the partitions. The summation of angles at the corner of all intersected partitions must be exactly 360 degrees, and this condition is satisfied only for triangles, quadrilaterals, and hexagons. To see this, if we have n-sided polygons tesselating the two dimensional plane, then look at the vertex V of one of the n-sided polygons. All the n-sided polygons at this point have to sum up a total angle of 360 degrees. But each n-sided polygon that meets at that point contributes either an angle of 180 degrees, if the point is along a side, or $\frac{180(n-2)}{n} degrees$. If there are k of the second kind, then either $180k\frac{(n-2)}{n} = 360$ if V is not on the side of any of n-sided polygons, or $180k\frac{(n-2)}{n} = 180$ if V is on the side of one of the n-sided polygons. When $n > 6$, neither of these can be true. Therefore, the largest number of edges that a regular polygon can have for a regular non-overlapping partitioning is 6. For $n = 5$, k is not found an integer. On the other hand, the number of tiles meeting at a single corner is 6 in triangular, 4 in rectangular, and 3 in hexagonal partitioning. This leaves only three basic tile shapes: triangles, rectangles, and hexagons.

B Total Boundary Length

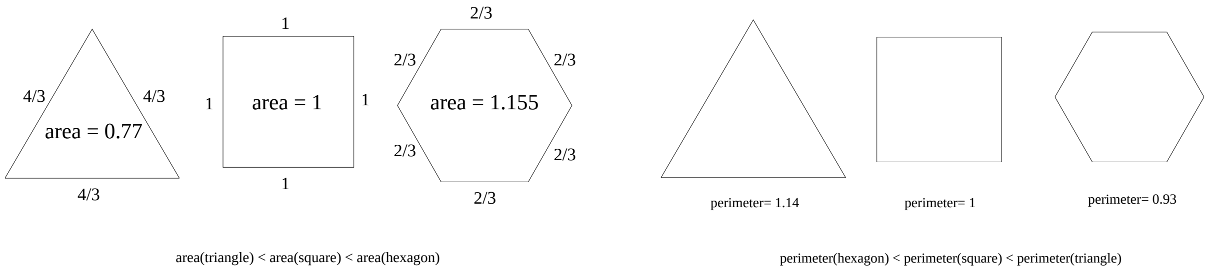
For example, in Figure 9(a) the data space is the unit square and is divided into sixteen partitions by using six line segments each of length 1, three vertical and three horizontal line segments not counting the boundary of the data space. The total length of the boundaries of this partitioning, excluding boundary of the data space, is 6 while the total boundary length in Figure 9(b) is 8.



(a) length=6 (b) length=8

Figure 9: Total Boundary Length

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 amount of boundaries that are used to tile a data space into n partitions is proportional to the number of partitions that are used and the perimeter of each partition. Recently, it has been observed that the total boundary of the partitions is a factor in determining the average number of page accesses for small circular queries [7]. 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



(a) Areas (fix perimeter=1 unit)

(b) Perimeters(fix area=1/16 units)

Figure 10: Area and perimeter comparison of three basic shapes

by circular queries of any size.

C Perimeter Comparison of the Three Basic Shapes

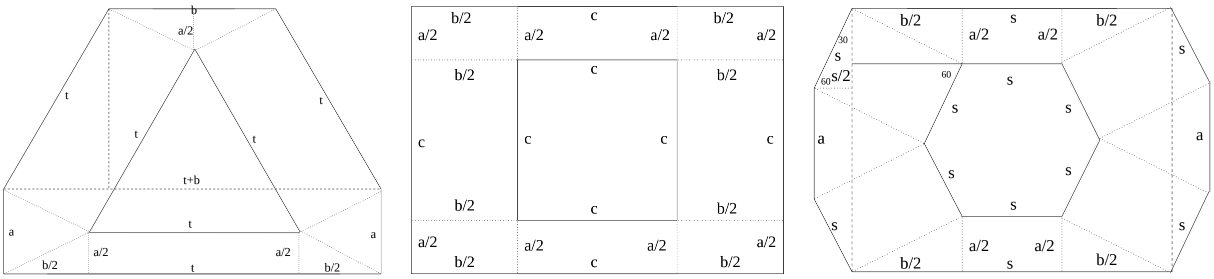
As mentioned in the paper, circles can not be used for non-overlapping partitioning of the data space. However, a circle has a very nice following property: it covers more area compared to its perimeter than any other shape. For a perimeter of 4 units, triangle covers $\frac{2\sqrt{3}}{3}$, square covers 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 proofs the optimality of regular hexagonal partitioning among all equal area partitioning techniques. A comparison of hexagonal partitioning with the three basic shapes is a good example of this observation. We compare the perimeter that is used by a triangle, a square, and a hexagon to cover 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 10). Therefore, to cover a fixed amount of 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.

D 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 11(b) shows the Minkowski sum region for each shape with equal area with respect to the rectilinear rectangular query (a, b), i.e., $MSum(triangle_t, rectangle_{a,b})$, $MSum(square_c, rectangle_{a,b})$, and $MSum(hexagon_s, rectangle_{a,b})$.

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



(a) Triangle-Rectangle (b) Square-Rectangle (c) Hexagon-Rectangle

Figure 11: Minkowski Sums w.r.t. Rectangles

$$Msum(square_c, rectangle_{a,b}) = \frac{3\sqrt{3}}{2}s^2 + (a+b)\sqrt{\frac{3\sqrt{3}}{2}}s + ab \tag{16}$$

The total area of the $Msum(hexagon_s, square_a)$ is the area of the rectangle inside the Msum plus the two trapezoid created on both sides of the Msum. Therefore,

$$MSum(hexagon_s, rect_{a,b}) = (s+b)(s\sqrt{3}+a) + \frac{(2a+s\sqrt{3})s}{2} \tag{17}$$

$$MSum(hexagon_s, rect_{a,b}) = \frac{3\sqrt{3}}{2}s^2 + 2as + bs\sqrt{3} + ab \tag{18}$$

Similarly,

$$MSum(triangle_t, rect_{a,b}) = \frac{3\sqrt{3}}{2}s^2 + \sqrt{6}as + \frac{3\sqrt{2}}{2}bs + ab \tag{19}$$

Comparing Equations 18 and 19, we have $MSum(hexagon_s, rect_{a,b}) < MSum(triangle_t, rect_{a,b})$. To compare Equations 16 and 18, 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 these equations, we find that $MSum(square_c, rectangle_{a,b}) < MSum(hexagon_s, rectangle_{a,b})$. Similar to the previous analysis, we conclude that $E_{square}(rectangle_{a,b}) < E_{hexagon}(rectangle_{a,b}) < E_{triangle}(rectangle_{a,b})$. For rectilinear rectangular range queries square grid partitioning outperforms hexagonal and triangular partitioning in terms of the expected number of page accesses.