ABSTRACT

Title of Dissertation: MULTI-DIMENSIONAL JOINS

Edwin H. Jacox, Doctor of Philosophy, 2007

Dissertation directed by: Professor Hanan Samet
Department of Computer Science

We present three novel algorithms for performing multi-dimensional joins and an in-depth survey and analysis of a low-dimensional spatial join. The first algorithm, the Iterative Spatial Join, performs a spatial join on low-dimensional data and is based on a plane-sweep technique. As we show analytically and experimentally, the Iterative Spatial Join performs well when internal memory is limited, compared to competing methods. This suggests that the Iterative Spatial Join would be useful for very large data sets or in situations where internal memory is a shared resource and is therefore limited, such as with today's database engines which share internal memory amongst several queries. Furthermore, the performance of the Iterative Spatial Join is predictable and has no parameters which need to be tuned, unlike other algorithms. The second algorithm, the *Quickjoin* algorithm, performs a higher-dimensional similarity join in which pairs of objects that lie within a certain distance ϵ of each other are reported. The Quickjoin algorithm overcomes drawbacks of competing methods, such as requiring embedding methods on the data first or using multi-dimensional indices, which limit the ability to discriminate between objects in each dimension, thereby degrading performance. A formal analysis is provided of the Quickjoin method, and experiments show that the Quickjoin method significantly outperforms competing methods. The third algorithm adapts incremental join techniques to improve the speed of calculating the Hausdorff distance, which is used in applications such as image matching, image analysis, and surface approximations. The nearest neighbor incremental join technique for indices that are based on hierarchical containment use a priority queue of index node pairs and bounds on the distance values between pairs, both of which need to modified in order to calculate the Hausdorff distance. Results of experiments are described that confirm the performance improvement. Finally, a survey is provided which instead of just summarizing the literature and presenting each technique in its entirety, describes distinct components of the different techniques, and each technique is decomposed into an overall framework for performing a spatial join.

MULTI-DIMENSIONAL JOINS

by

Edwin H. Jacox

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Acknowledgments

I would like to thank my colleagues at the University of Maryland for providing a rich intellectual environment, especially, Jagan Sankaranarayanan, Houman Alborzi , and Gísli Hjaltason. I appreciate the insight and discussions that David Mount provided regarding the theoretical work. Before coming to the University of Maryland, my advisors from the University of Michigan, Al Ward and Bill Birmingham provided me with a solid foundation on which to build upon.

The continuing support of my friends and family has been and always will be a source of strength for me. I am especially thankful to my parents, Edwin and Rose Ann, and Rob, Kristina, and Julie, as well as many aunts, uncles, and cousins. Moreover, without Ada's encouragement, I would never have started this long journey. I am thankful to John Page for continuing to be a friend and mentor, and I am much obliged to my friends for their continuing support, in this last year, especially to Beth, Jon, Angela, Hui Ling, Eric, Joy, Jim, and Nicole.

Thank you Alexandra and Julie.

Most of all, thank you Hanan for giving me the opportunity to work with and learn from a brilliant professor. I very much appreciate all that you have done.

This page intentionally left blank.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Given two datasets, a relational join finds all pairs of tuples, one from each set, that satisfy a given join condition. This fundamental database operation also applies to datasets in more than one dimension, and requires techniques beyond the traditional relational join algorithms, such as the sort-merge join and the hash join [56]. In two and three dimensions, the datasets most notably involve spatial data representing geographic objects, and the join operation is termed a *spatial join*. For example, a spatial join answers a query such as *find all of the rural areas that are below sea level*, given an elevation map and a land use map. In higher dimensions, the *similarity join* finds pairs satisfying the join condition (typically a distance function) on datasets such as feature vectors extracted from, for example, similar images or similar sub-sequences. The similarity join is a typical data-mining operation that has applications in many domains, such as medical imaging [115], multimedia [60], and GIS [169]. The operation is used to find association rules [114], to analyze time series, such as stock histories [5, 6], and can also be used to identify data clusters [30].

Additionally, a form of a multi-dimensional join is needed to calculate the Hausdorff distance, which is used as a measure for matching images in applications such as pattern recognition, computer vision, image analysis, and surface approximation. Specifically, in pattern recognition and computer vision applications, the Hausdorff distance is used to check if a match is found for a target image within a desired image [167, 168] or to determine the similarity of two shapes [94]. As noted by Rucklidge [167], "the Hausdorff distance is reliable even when the image contains multiple objects, noise, spurious features, and occlusions." The Hausdorff distance is also used in image processing for contour or line detection [209, 210], image registration [137] (for example, stitching multiple images into a panoramic view), stereo matching (recovering 3-d information from two 2-d images of the same scene) [108], and indexing images [208]. These techniques have been applied in several diverse domains, such as segmentation in medical imaging [42, 159] as well as in a method to find matching astronomical images [157]. The Hausdorff distance has also been used as an error measure in creating a triangular mesh for approximating a surface [10].

We first focus on the spatial join. As it is a fundamental database operation, applying to not only geographic information systems, but also any domain representing two or three-dimensional data, a large body of diverse literature exists on the topic. Chapter 2 presents an in-depth survey and analysis of spatial joins that not only presents a survey of the literature on spatial joins, but also extracts algorithms and techniques from the literature, and presents a coherent description of the state of the art in the design of spatial join algorithms. The remainder of this chapter introduces the concepts behind three novel algorithms, one for performing a spatial join on unindexed data, one for performing a similarity join, and another for calculating the Hausdorff distance using incremental distance join techniques.

These three algorithms are presented in detail in Chapters 3, 4, and 5, respectively. Conclusions are drawn in Chapter 6.

1.1 Spatial Joins On Unindexed Datasets

With respect to spatial joins, it is generally an axiom that indexing is crucial for efficiently processing queries on multidimensional data, and methods for processing a spatial join using indices are discussed in Sections 2.3.1 and 2.3.2. However, there are many situations where indexing does not necessarily pay off since the time needed to actually build the index prior to the performance of the operation is an issue. For example, consider the spatial spreadsheet [97]. In this case, the output of a spatial operation serves as input to another spatial operation. If the output is not reused several times, then it might not be worthwhile to expend the work to build an index as the subsequent operation's execution time might not be dramatically reduced by the presence of an index. This is especially true for spatial joins that make use of indices that take a long time to build (e.g., an R*-tree [25]).

Chapter 3 presents a novel algorithm, the *Iterative Spatial Join*, for performing spatial joins without the use of an index on the data. Other approaches to performing spatial joins that claim not to use an index apply a partitioning scheme to the underlying space (e.g., [19, 116, 124, 155], discussed in Chapter 2.3.3). The effect of the partitioning is the creation of the analog of either an inverted file (e.g., [19]) or a fixed-grid (e.g., [116, 124, 155]). However, the inverted list (e.g., [113]) and the fixed-grid (e.g., [29, 113]) are actually the building blocks of all indexing methods in that, in essence, their use results in sorting the objects along one or all of their attributes, respectively, into bins. Thus, these methods actually do use a form of an index.

As shown in Chapter 3, the Iterative Spatial Join method does not use any index. Furthermore, the difference between the Iterative Spatial Join and the partitioning methods is similar to the difference between the sort-merge join and the hash-based joins for the traditional one-dimensional join [71]. The hash-based methods partition the data and then join the partitions in a divide-and-conquer approach, which is analogous to the way the partitioning methods perform the spatial join. The sort-merge join, on the other hand, sorts the data first, and then merges the two datasets by sweeping a point across the data. If the data does not contain duplicate values and the join condition is equality, then each item can be read one at a time (or a block a time), compared to the current value in the other dataset, and then immediately purged from internal memory. However, if the data does contain duplicate values, then each duplicate must be kept in internal memory until new values appear in both datasets so that they are correctly matched with the corresponding values in the other dataset. In other words, for the equality join, any value that intersects the current value (point value for integers), needs to be kept in internal memory. Since the data is sorted, the point is *swept* across the data. This is similar to the two-dimensional plane-sweep method [160]. In particular, for the plane-sweep method, the data is sorted along one of the dimensions. A sweep line, rather than a point, is then swept over the data. Any object intersecting the line is termed *active* and needs to be kept in internal memory to be compared to later objects. However, since we are working in two dimensions, there is much more data intersecting the sweep line, and thus the intersecting (i.e., active) data is not likely to fit in internal memory for larger datasets. This concept forms the basis for the Iterative Spatial Join, which addresses the problem of overflowing internal memory. In particular, when internal memory overflows, excess data (or a reference to the data) is written to a file and the Iterative Spatial Join makes additional passes on the data.

1.2 Similarity Joins

For the similarity join, similarity for a pair of objects means that they are within a certain distance, ϵ, of each other. The data can be of any type, as long as a distance function exists that returns a distance between any pair of objects. For many domains, the data is often represented as points in a multi-dimensional space or as feature vectors. Other related operations are k-nearest neighbor joins [206], which find the k most similar objects for each object, approximate spatial joins [149], which find the most similar pairs within a given amount of time, and all-nearest-neighbor queries [212], which find the nearest object for each object in the set.

Currently, existing similarity join techniques are only applicable to multi-dimensional point (vector) data. If the data is not in a vector format, then embedding methods [62, 89, 121, 202] are usually used to transform the data into a vector format. Furthermore, most existing similarity join algorithms, discussed in Section 4.1, use some form of a multi-dimensional index, which often rely on picking particular dimensions as either the primary sort dimensions or the dimensions to index. For example, the ϵ-kdb tree [177] splits the data along one dimension first, then another, and so forth. This presents several problems. First, a good dimension must be found. In particular, if an inappropriate dimension is chosen, then performance could be poor. For instance, a bad dimension is one in which the range of values in the dimension is nearly ϵ. In this case, every object is similar in that dimension, thereby making it useless to the index and leading to wasted processing and storage.

Other methods try to index all of the dimensions at once as in a grid approach [51, 103, 117]. However, with higher dimensions a problem arises in that it becomes very likely that most data points will have features that are near the grid boundaries. For instance, consider a multi-dimensional uniform grid where the data values range from 0 to 1 in each dimension, and each dimension is split in two with the grid boundary at 0.5. For an ϵ value of 0.1, the probability that a random point lies within ϵ of the grid boundary in a single dimension is 0.2. Therefore, on average, for 100 dimensions, the point would be near 20 boundaries. In a simple grid index, similar to the one described in [103], the join is performed by comparing overlapping cells between the two grid-indexed datasets. In this approach, points within ϵ of a grid boundary need to be replicated into the cells on both sides of the grid boundary. If a point is near two boundaries, then it needs to be replicated into four cells, and so forth. If a point is near 20 boundaries, then the point would need to be replicated into 2^{20} grid cells, making this approach unsuitable for higher dimensions. Two of the grid methods [51, 117], described in Section 4.1.2, use an approach that requires little or no replication, but performance still degrades significantly for higher dimensions, as shown experimentally in Section 4.4. Additionally, as we quote in Section 4.1.2, the creators of one of the methods [51] concede that this approach is unsuitable for higher-dimension similarity joins.

Another deficiency of multi-dimensional indices is that they are built efficiently for only one distance function or one embedding of the data. In the past, this has not been an issue since the most commonly used distance function is one of the Minkowski metrics, such as the Euclidean distance (L_2-norm). As pointed out by Aggarwal [4], the L_2-norm is not necessarily relevant to many emerging applications involving high-dimensional data. Furthermore, the distance function might change for a particular application based on user preferences. In these cases, a dimension that is primary for one distance function, might not be so for another distance function. For instance, with wine data [82, 62], a user could decide that a particular feature of the wine data is irrelevant, and set the weight of that dimension to zero. In the worst case, every dimension that is used by the index could be assigned a weight of zero, in which case the index is useless.

From a theoretical perspective, the ability to discriminate between objects in any particular dimension becomes more limited as the number of dimensions that make up the space increases. To see this, consider a data space in d dimensions where the data is uniformly distributed across all dimensions with a range of values of 0 to 1 in each dimension. Using an unweighted L_2-norm, the maximum distance between any pair of objects is $\sqrt{d}$, where the distance between the objects in each dimension is one. Therefore, the maximum ϵ value is $\sqrt{d}$. However, for multi-dimensional indices used in similarity joins, the indices are built by creating branches that split the objects on one dimension at a time, rendering the multi-dimensional index useless for any ϵ value over one, or even near one, since any search will return all of the objects. For example, in 100 dimensions, two objects could be a distance of 10 apart, yet any ϵ value over 1 will cause every search of the index to return all objects. Since all objects are returned for each search of the index, which is performed for each object, the result is extremely poor performance. Therefore, in lower dimensions, the multi-dimensional index approaches will work fine, but for higher dimensions, especially very high dimensions, the performance of the multi-dimensional index methods will degrade.

Chapter 4 introduces a novel distance-based algorithm that overcomes these problems. The algorithm, termed *Quickjoin*, uses concepts similar to those used in distance-based indexing methods [43, 85] in that only the distance function is used, rather than relying on a particular dimension. In other words, the data does not need to be in vector format (multi-dimensional points). Additionally, the method does not build an index, so it is able to adapt to changing distance functions. The only requirement is that the triangle equality hold, which is true in many applications.

1.3 The Hausdorff Distance

The Hausdorff distance is a measure of the maximum of the minimum distance between two sets of points, the surfaces of two objects, or any two sets of objects. For instance, as shown in Figure 1.1, a low Hausdorff distance value corresponds to

(a) H=0.1872 (b) H=0.1370

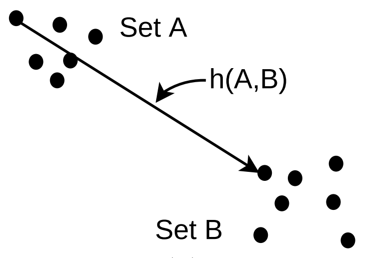
Figure 1.1: As a matching example, the pairs of lake images with the lowest Hausdorff distance from a collection of lakes are shown.

similar images. In this example, the images are polygons corresponding to lakes from the dataset of the Ontario Ministry of Northern Development and Mines (MNDM) [142] represented as collections of vertices, which are pairs in two dimensions. In this case, to match images, each image was normalized so that its coordinate values range from 0 to 1 in each axis, which adjusts the images for translation, but not for rotation. For this reason, the similar images have similar rotation.

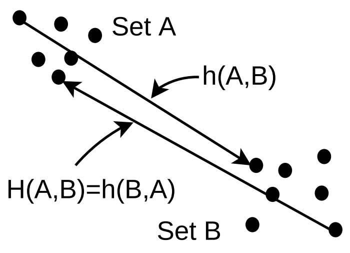
The Hausdorff distance is defined as the maximum of the minimum distance between an element in a set A, to the nearest element in a set B, as shown in Figure 1.2a. Formally the Hausdorff distance is given in Equation 1.1, where $d(a, b)$ is some distance measure (metric) between elements a and b of sets A and B, respectively.

$$h(A, B) = max_{a \in A} min_{b \in B} d(a, b) \tag{1.1}$$

This definition of the Hausdorff distance, referred to as a *directed Hausdorff distance*, $h(A, B)$, is not symmetric, as illustrated in Figure 1.2b. The symmetric version of the Hausdorff distance, $H(A, B)$ takes the maximum of the two directed distances, as shown in Equation 1.2.



(a)



(b)

Figure 1.2: (a) The directed Hausdorff distance, $h(A, B)$, is the maximum minimum distance between two sets, and (b) is not symmetric, though the Hausdorff distance, $H(A, B)$ is symmetric since it is defined as the maximum of $h(A, B)$ and $h(B, A)$.

$$H(A, B) = max\,[h(A, B), h(B, A)] \tag{1.2}$$

The traditional approach [8] to calculate the Hausdorff distance uses a Voronoi diagram, where the sites are the vertices of the polygons. In Chapter 5, we show that incremental distance join concepts [83, 179, 180, 206] can be used to improve the performance of this calculation process for datasets that can be indexed by hierarchical containment spatial indices, such as the R-tree [76].

Chapter 2

Survey of Spatial Joins

This chapter presents an in-depth survey and analysis of spatial joins. A large body of diverse literature exists on the topic of spatial joins. The goal of this chapter is not only to survey the literature on spatial joins, but also to extract algorithms and techniques from the literature and to present a coherent description of the state of the art in the design of spatial join algorithms. Frequently, an article presents a complete framework for performing a spatial join. Instead of summarizing each complete framework individually, we decompose them into components in two ways. First, if several methods are similar, then a common algorithm is extracted from the frameworks to show specifically how each framework differs from the others. For instance, there exist several methods for performing a spatial join on R-trees [76], each using a hierarchical traversal method. From these different algorithms, we create a generic hierarchical traversal algorithm and show how each method slightly varies the generic algorithm (see Section 2.3.1.1). Thus, each method is presented in a simpler manner that allows it to be more thoroughly compared and contrasted with similar algorithms. By doing so, the strengths and weaknesses of each competing algorithm become more apparent. The various components are tabulated in Table 2.1 along with the sections in which they are discussed. The second approach to decomposing the various methods is to extract common issues from each

Table 2.1: Components of spatial join algorithms.

Internal Memory Methods	2.2.1 Nested-Loop Join [136] 2.2.2 Indexed Nested-Loop Join [56] 2.2.3 Plane Sweep [19, 160] 2.2.4 Z-Order [14, 143]
Section 2.3.1 External Memory Methods (Both Datasets Indexed)	2.3.1.1 Hierarchical Traversal [38, 72, 91, 109] 2.3.1.2 Non-Hierarchical Methods [78, 110] 2.3.1.3 Multi-Dimensional Point Methods [181]
Section 2.3.2 External Memory Methods (One Dataset Not Indexed)	2.3.2.1 Construct a Second Index [122] 2.3.2.2 The Index as Partitioned Data [192, 132] 2.3.2.3 The Index as Sorted Data [18, 74]
Section 2.3.3 External Memory Methods (Neither Dataset Indexed)	2.3.3.1 Generic Partitioning Algorithm 2.3.4 Grid Partitioning [155, 213] 2.3.5 Strip Partitioning [19] 2.3.6 Size Partitioning [116, 19] 2.3.7 Data Partitioning [123, 124]
Section 2.5 Refinement	2.5.1 Ordering Candidate Pairs [1] 2.5.2 Polygon Intersection Test [160, 37] 2.5.3 Alternate Intersection Test [37]

Table 2.2: Spatial join issues.

Section 2.1 Spatial Join Basics	2.1.1 Design Parameters 2.1.2 Minimum Bounding Rectangles 2.1.3 Linear Orderings
Processing Issues	2.3.1.4 Joining Data Nodes
Partitioning Issues	2.3.3.2 Determining the Number of Partitions 2.3.3.3 Repartitioning 2.3.3.4 Avoiding Duplicate Results
Section 2.7 Selectivity Estimation	Uniform Dataset Estimates [13] Non-Uniform Dataset Estimates [27, 48, 63, 131]
Section 2.4 Alternate Filtering Techniques	2.4.1 False Hit Filtering [36, 116, 198, 218] 2.4.2 True Hit Filtering [34] 2.4.3 Non-Blocking Filtering [126]

Table 2.3: Specialized spatial joins.

Section 2.6.1 Multiway Spatial Joins	2.6.1.1 Multiway Indexed Nested-Loop [133, 130, 150] 2.6.1.2 Multiway Hierarchical Traversal [133, 130, 150]
Section 2.6.2 Parallel Spatial Joins	Parallel Hierarchical Traversal [39] Parallel Grid Partitioning Methods [126, 156, 213] Hypercube Spatial Joins [87]
Section 2.6.3 Distributed Spatial Joins	Distributed Filter and Refine [2, 128]

and address these issues in separate sections. For example, many of the spatial join methods for handling unindexed data must deal with the issue of removing duplicate results from the different stages of spatial join processing. Rather than separately show how each framework handles duplicate results, different techniques for handling duplicate results are described in a separate section (Section 2.3.3.4). The sections dealing with issues that arise in algorithms are tabulated in Table 2.2. Furthermore, spatial joins for specialized environments are discussed in separate sections, as tabulated in Table 2.3.

The rest of the chapter is organized as follows. Section 2.1 defines the spatial join operation and discusses design parameters that influence the performance of a spatial join as well as describing the following two concepts that are important to many spatial join algorithms: the *minimum bounding rectangle* (also known as a *minimum bounding box*) and *linear orderings*. Typically, a spatial join is performed in two stages: the *filter* stage in which complicated polygonal objects are approximated by rectangles and the *refinement* stage which removes any results produced during the filtering stage that do not satisfy the join condition [145]. Section 2.2 describes internal memory filtering techniques, while Section 2.3 describes external memory filtering techniques. Section 2.4 presents extended or alternate filtering techniques, while Section 2.5 discusses the refinement phase. Section 2.6 explains how spatial joins are handled in specialized situations, such as in parallel architectures, while Section 2.7 discusses selectivity estimation.

2.1 Spatial Join Basics

Given two datasets of multi-dimensional objects in Euclidean space, a spatial join finds all pairs of objects satisfying a given relation between the objects that involves the values of their spatial components, such as intersection. For example, a spatial join answers such queries as *find all of the rural areas that are below sea level*, given an elevation map and a land use map [198]. To illustrate the concept further, a simplified version of a spatial join is as follows: given two sets of rectangles, R and S, find all of the pairs of intersecting rectangles between the two sets, that is, for each rectangle r in dataset R, find each intersecting rectangle, s, from dataset S, as illustrated in Figure 2.1. The general spatial join problem, also known as a *spatial overlay join*, extends the simplified version in several ways:

1. The datasets can be objects other than rectangles such as points, segments, or polygons [1].

[1]An extensive amount of research has also examined the related operation, the segment join [21, 40, 44, 127, 193], and the more general overlay operation which also addresses the effects of the result of the spatial join on the way in which the combined attributes of the join are handled [195, 196, 197]

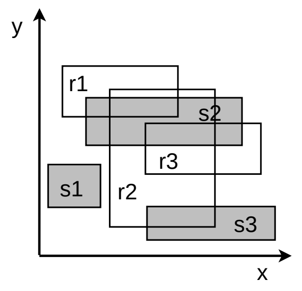


Figure 2.1: A spatial join to find the intersecting objects of datasets R, consisting of objects $r1$, $r2$, and $r3$, and S, consisting of objects $s1$, $s2$, and $s3$, will report the intersection of objects $r1/s2$, $r2/s2$, $r2/s3$ and $r3/s2$.

2. The datasets might have more than two dimensions [2].

3. The relationship between pairs of objects can be any relation between the objects that involves the values of the spatial components, such as intersection, nearness, enclosure, or a directional relation (for example, find all pairs of objects such that r is northwest of s [216]).

4. There might be more than two datasets in the relation (a *multiway spatial join*) or only one dataset (a *self spatial join*).

The problem of spatial join has been the subject of much attention in fields other than spatial databases. In particular, its solutions make use of the same principles as interference detection in robotics applications (for example, [70]), game programming (for example, [189]), and design rule checking in VLSI applications (for example, [164]). These topics are beyond the scope of this review, but for more details, the interested reader should consult texts such as [172].

[2] A one dimensional version of the spatial join would be an interval join [57].

Spatial joins are distinguished from a standard relational join [136] in that the join condition involves the multi-dimensional spatial attribute of the joined relation. This property prevents the use of the more sophisticated relational join algorithms. For instance, because the data objects are multi-dimensional, there is no ordering of the data that preserves proximity. Relational join techniques that rely on sorting the data, such as the sort-merge join [136], work because neighboring objects (those with the next higher and lower value) are adjacent to each other in the ordering. However, in more than one dimension, the data can not be sorted so that this property holds for all directions and dimensions. For example, in two dimensions, the left and right neighbors can be adjacent to an object in an ordering, but then the top and bottom neighbors will need to go elsewhere in the order (see Section 2.1.3 for a further discussion of multi-dimensional orderings).

Other relational join techniques are also inapplicable because the data objects might have extent. For example, equijoin techniques [136] (for example, hash joins), will not work with spatial data because they rely on grouping objects with the same value, which is not possible when the objects have extent. This is the same reason that equijoin techniques will not work with intervals (extent in one dimension) or inequalities. As an example, for a one-dimensional hash join on datasets R and S, a group of equal-valued objects, say G, is formed from dataset R and placed in a bucket. If any object g in bucket G satisfies the relation with an object from dataset S, then so does every object from dataset G. This property does not hold for objects with extent, such as a rectangle, because the objects can overlap each other and a disjoint grouping might not exist. In fact, an object from dataset R could potentially intersect every object from dataset S. Because of these two factors (that is, failure to satisfy proximity preservation and extent) relational join algorithms cannot be used directly to perform a spatial join.

The computational geometry approach to solving the simplified spatial join (a two-set rectangle intersection) is to use a plane-sweep technique [160] (see Section 2.2.3). In order to use the plane-sweep method for a general spatial join, two problems must be overcome: the objects are not necessarily rectangles and there might not be sufficient internal memory for the plane-sweep algorithm. Furthermore, calculating whether two complex objects satisfy the join condition, such as intersection, can be an expensive operation, and performing as few of these operations as possible improves overall performance. To overcome these problems, a spatial join is typically performed using a two stage filter-and-refine approach [145].

In the filter-and-refine approach, the spatial join is first solved using approximations of the objects in the filtering stage and any incorrect results due to the approximations are removed in the refinement stage using the full objects [3]. In the filtering stage, objects are typically approximated using *minimum bounding rectangles* (see Section 2.1.2), hereafter referred to as MBRs, which require less storage space than the full object, resulting in faster processing and less expensive I/O operations [4]. For example, GIS objects might be polygons, each consisting of thousands,

[3]While the filter and refine stages can be considered two phases of one technique, Park et al. [154] propose separating the filter and refinement steps for query optimization so that each stage can be combined with non-spatial queries.

[4]Other approximations also can be used (see Section 2.4).

or even millions, of points. Reading these objects in and out of memory could easily be the dominant cost of performing a spatial join, depending upon the available amount of internal memory and the ratio of I/O to CPU performance, whereas a filter-and-refine approach alleviates this problem. Furthermore, a spatial join on rectangles presents a more tractable problem. For smaller datasets, the filtering stage of the spatial join can be solved using internal memory techniques, which are described in Section 2.2. For larger datasets, external (secondary) memory techniques are required for the filtering stage, which are described in Section 2.3.

The output of the filtering stage is a list of all pairs of objects whose approximations satisfy the join condition, which is referred to as the *candidate set*, and is typically represented by pairs of object ids. The candidate set includes all of the desired pairs, those whose full objects satisfy the join condition, but also includes pairs whose approximations satisfy the join condition, but whose full objects do not. The extra pairs appear because of the inaccuracy of the object approximations (see Section 2.1.2). The purpose of the refinement stage is to remove the undesired pairs using the full objects, producing the final list of object pairs that satisfy the given join condition. Refinement techniques are described in Section 2.5.

As mentioned above, the dominant cost of a spatial join with very large objects can be the I/O cost of reading the large objects, depending upon the amount of internal memory and the ratio of I/O to CPU performance. Early filtering techniques were dominated by I/O costs [38]. Later techniques have improved I/O performance so that it is no longer an axiom that I/O costs dominate the CPU costs [155]. Even though filtering reduces the I/O costs, reading large objects can still be the major cost of the refinement stage, which is generally more expensive than the filtering stage [155]. Furthermore, while the performance improvement from using a filter-and-refine approach might be obvious for very large objects, it remains an open question as to whether it is the best approach for smaller, simpler objects. As an example of an alternative approach, Zhu et al. [214, 215] have proposed methods for extending the plane-sweep algorithm (Section 2.2.3) to trapezoids and recti-linear polygons, thereby avoiding the need for the filter-and-refine approach for objects having such shapes.

Throughout the review of the spatial join techniques, we do not discuss experimental results. Most of the methods described in this survey were shown to outperform some other method. Unfortunately, we found it difficult to compare methods using only the literature since the techniques are compared with one or no other technique, and the implementations of the techniques can vary dramatically, which has a large impact on the experimental results. Furthermore, the variety of computer hardware, software and networks used make it difficult to compare results between methods. For these reasons, we do not discuss most experimental results.

Also, to simplify the discussion of the techniques, it is assumed that the data is two dimensional and that we are interested in determining pairs of intersecting objects. Both of these assumptions are common in the literature. The two-dimensional assumption is made because the applications of these techniques to higher dimensional data has not been extensively addressed in the literature and many of the techniques presented might not work or might not perform well in higher dimensions. The intersection assumption is made only to simplify the discussion. We believe that this assumption does not effect the generality of the algorithms. For example, a nearness relation can easily be calculated by extending the size of the MBRs so that nearness is calculated by an intersection test [117]. However, for some predicates, such as a directional predicate, the algorithms need to be modified appropriately. For instance Zhu et al. [216] use a modified plane-sweep algorithm (see Section 2.2.3) to search for all objects in the desired direction for a directional predicate. When appropriate, a generic join condition is used, rather than intersection.

Furthermore, although many spatial join techniques depend on spatial indices, the discussion of spatial indices is left to other work [68, 170]. Knowledge of these structures can be crucial to a deeper understanding of many of the techniques for processing spatial data. Where appropriate, these index structures are described, but in general, the algorithms are presented in such a way that little or no knowledge of the underlying spatial indices is required.

The remainder of this section discusses issues that are fundamental to the design of spatial join algorithms. Section 2.1.1 discusses various design considerations and parameters that influence the performance of spatial join algorithms. Sections 2.1.2 and 2.1.3 review MBRs and linear orderings, respectively, which are concepts that are fundamental to many spatial join techniques.

2.1.1 Design Considerations and Parameters

Many factors contribute to the performance of a spatial join and influence the design of algorithms. The foremost factor, of course, is the processor speed and I/O performance, and in particular, the ratio of these two factors. Early spatial joins algorithms were constrained by I/O, which dominated CPU time, and the focus of improvements was on minimizing the amount of data that needed to be read from and written to external memory. As spatial join algorithms improved, experiments showed that CPU time accounted for an equal share of performance and that the algorithms were no longer I/O dominated [38]. In addition, ever increasing amounts of internal memory allow larger portions of the data to reside in memory, which also improves the performance of the spatial join. Today, algorithms need to account for both CPU performance and I/O performance. These two factors can be balanced somewhat by tuning page sizes and buffer sizes (the amount of internal memory available to the algorithm), two factors which also play an important role in performance. However, as processor, I/O speeds, and internal memory sizes continue to improve, algorithms need to account for these factors and thus tuning will always be necessary for the best performance.

The characteristics of the datasets and whether the datasets are indexed are also major influences on performance. The dataset sizes obviously effect overall performance, but a more important issue is whether the dataset fits into the available internal memory. If the entire dataset does fit in internal memory, then the spatial join can be done entirely in memory (Section 2.2) [5], which can be signifi-

[5]If the plane-sweep technique is used (Section 2.2.3), then the dataset can be processed in

cantly faster than using external memory methods (Section 2.3). One of the most confounding factors for spatial join design is the distribution of data. Algorithms for uniformly distributed datasets are easy to develop, but the development of algorithms for handling skewed datasets is significantly more complicated. A poorly designed algorithm can thrash with skewed datasets by repeatedly reading the same data in and out of external memory, which severely degrades performance. These factors are mitigated if the data is indexed appropriately. If a dataset is indexed, then in general, algorithms that use the index will be faster than those that do not. Section 2.3 classifies spatial join algorithms by whether they assume that both datasets are indexed (Section 2.3.1), only one dataset is indexed (Section 2.3.2), or neither of the datasets are indexed (Section 2.3.3).

How the data is stored is another factor that contributes to the design of spatial joins. Vectors (a list of vertices) are commonly used to store polygons, but raster approaches are also used [143]. The choice of storage method for the full object mostly effects the complexity of the object intersection test during the refinement stage, since an approximation of the full object is used during the filtering stage. This work only discusses refinement techniques for the more common vector representation. During the filtering stage, an object is represented by an approximation and an object id or a pointer is used to access the full object. An MBR is generally chosen as the approximation, but other approximations can also be used (see Section 2.4).

internal memory even if its total size exceeds the size of the internal memory provided that the data is already sorted and that the active set fits into internal memory.

The environment in which the algorithm is executed also plays a role in the design of spatial join algorithms. In a demand-driven pipelined system [71], which is typical for a DBMS, each stage of the spatial join algorithm needs to output results continuously in order for the pipeline to run efficiently. In this case, each stage is said to be *non-blocking* because the next stage does not need to wait for results. Unfortunately, filtering methods that sort or partition the data are blocking, although there is a method to produce some results earlier (see Section 2.4.3). Also, specialized algorithms can be used to improve the performance of multiway spatial joins, and modified algorithms are required to perform spatial joins that run in parallel environments and distributed environments (see Section 2.6).

2.1.2 Minimum Bounding Rectangles and Approximations

For the filtering stage, most algorithms use a minimum bounding rectangle (MBR) to approximate the full object, though other approximations might be used instead or as a secondary filter (Section 2.4). An MBR of an object is the smallest enclosing rectangle whose sides are parallel to the axes of the space (axis-aligned), as shown in Figure 2.2a. MBRs are preferred over the full object because they require less memory and intersections between MBRs are easier to calculate. Objects, especially in GIS applications, can be very large, requiring many points or lines to represent a polygon, and for large datasets, which are also typical in GIS applications, reading thousands, millions, or more of these objects from external memory and performing intersection tests on them can be extremely expensive in terms of

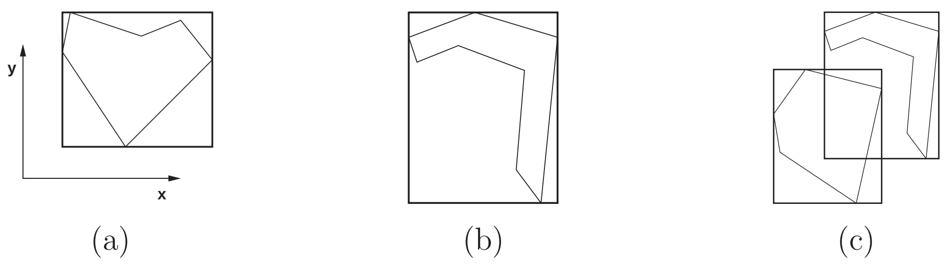


Figure 2.2: (a) A minimum bounding rectangle (MBR) is the smallest rectangle that fully encloses an object and whose sides are parallel to the axes. (b) The area of an MBR might be significantly larger than the area of the enclosed object. The extra area is referred to as *dead space*. (c) Two MBRs might intersect even though the objects that they enclose do not intersect.

I/O performance. Instead, if MBRs are used in the filtering stage, the MBRs can be read from external memory faster than the full object and the intersection tests can be performed faster. Unfortunately, using MBRs, or any approximation, will produce some wrong answers. As shown in Figure 2.2b, an object might only occupy a fraction of its MBR, leaving a portion of *dead space*. Two MBRs might intersect, but the objects they represent might not intersect, as shown in Figure 2.2c. This result is referred to as a *false hit*, whereas the result is termed a *true hit* if the MBRs intersect and the objects they represent also intersect.

Before performing a spatial join, the MBRs for the full objects must be calculated. If the data is indexed using a spatial indexing method [68, 170], then typically the MBRs exist already. If they do not, then a scan of the full dataset is required to create the MBRs. Forming the MBR of an object simply involves checking each corner point of the object, which is an $O(n)$ operation for a polygonal object with n vertices.

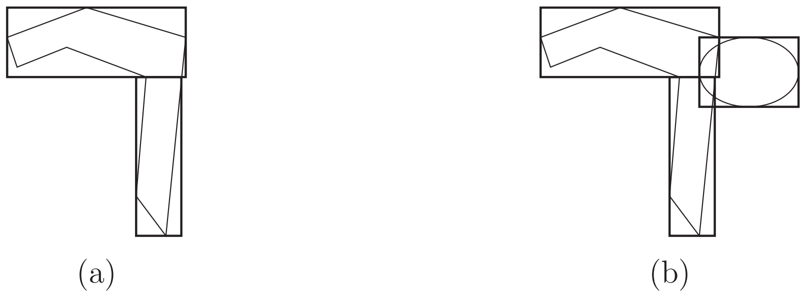


Figure 2.3: (a) In order to reduce dead space, an object can be approximated by two disjoint rectangles. (b) However, both rectangles might intersect a second object, thereby producing duplicate results.

Use of the filter and refine approach for spatial joins was first introduced by Orenstein [145]. Orenstein was concerned that a poor approximation would degrade performance [144] and experimented with using a set of disjoint rectangles to approximate each object. For example, to use this representation, the object in Figure 2.2b is decomposed into the two MBRs shown in Figure 2.3a, improving the approximation by reducing the dead space, but also increasing the size of the dataset. While this approach improves the accuracy of the filter stage, it also creates the need for an extra step after the filtering stage to remove duplicates from the candidate set. As shown in Figure 2.3, both pieces of the decomposed object in Figure 2.3a might intersect the same object, as shown in Figure 2.3b. Both of these intersections create a candidate pair. The duplicate results generally need to be removed for most applications and this typically should be done before the more costly refinement stage in order to avoid extra processing (see Section 2.3.3.4 for a discussion of duplicate removal techniques).

Nevertheless, most algorithms use one MBR, rather than approximating an object by a set of rectangles, and rely on the refinement stage to efficiently remove false hits. However, many algorithms intentionally duplicate objects. For instance, if an algorithm creates a disjoint partition of the objects in a divide-and-conquer approach, as is done with a grid partitioning approach (see Section 2.3.4), then each object will appear in each partition it overlaps. Similarly, some spatial indices that can be used to perform a spatial join use disjoint nodes and the objects again are copied into each node they overlap (for example, the R$^+$-tree [175]). In both cases, duplicate removal (or avoidance) techniques are required (see Section 2.3.3.4).

2.1.3 Linear Orderings

A linear order [99, 170] creates a total order on multi-dimensional objects. In other words, a linear order is a traversal of all of the objects, as shown in Figure 2.4. Linear orderings play an important role in many spatial join techniques, in a similar way that sorted orders (a one-dimensional linear ordering) play an important role in creating efficient algorithms for relational joins (for example, the sort-merge join [136]). The benefit of a sorted order in one dimension is that neighboring objects (close in value) are next to each other in the sorted order, leading to algorithms such as the sort-merge join [136]. In more than one dimension, no natural linear order exists and spatially neighboring objects will not necessarily be close in the linear order. For example, in Figure 2.4a, neighboring objects a and b are next to each other in the order indicated by the arrows, but in Figure 2.4b, they are widely separated. Nevertheless, because linear orders keep some of the neighboring objects

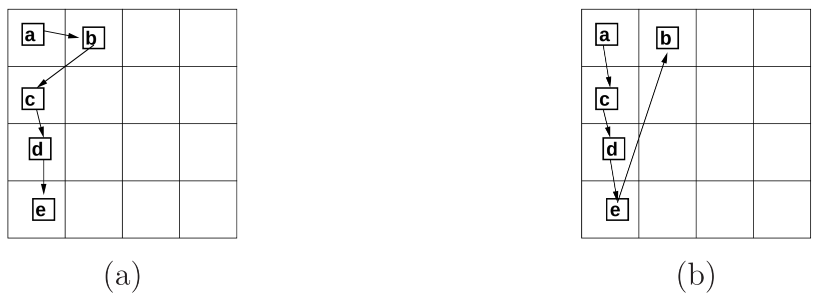
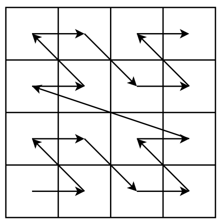


Figure 2.4: (a) A linear order, where object a's neighbors, b and c are nearby. (b) A linear order in which one of object a's neighbors, c, is nearby, but the other neighbor, b, is not.

near each other in the order, they can be useful in spatial join techniques.

Linear orders that keep neighboring objects closer in the order, on average, such as the Z-order or the Peano-Hilbert order, tend to be more useful for spatial join algorithms. The Z-order (also known as a Peano or Morton order), shown in Figure 2.5a, and the Peano-Hilbert order, shown in Figure 2.5b, traverse the grid in a pattern that helps to preserve locality. They are also known as space-filling curves (see [172] for other such curves). Both of these linear orders first order objects in a block before moving to the next block. For example, the Z-order is a traversal through a regular grid using a 'Z' pattern, as shown in Figure 2.6a. If there are more than four cells in the grid, then each top-level block is fully traversed before moving to the next block. In Figure 2.6b, block A from Figure 2.6a is traversed in a 'Z' pattern before moving to block B. This pattern is repeated at finer levels, where a block at any level is fully traversed before moving to the next block. In this way, a linear order is imposed on the cells. The Peano-Hilbert order is similar, as shown in Figure 2.5b, though each block is traversed in a rotation that might be clockwise


(a)

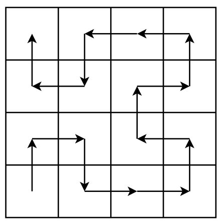
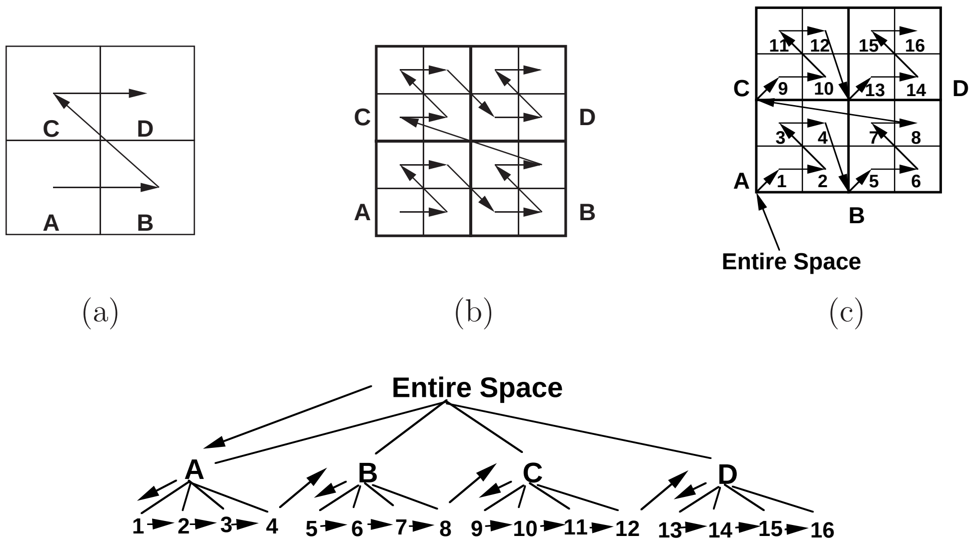

(b)

Figure 2.5: (a) Z-Order (Peano order) and (b) Peano-Hilbert order.

or counter-clockwise, avoiding the large jumps between the constituent grid cells of a Z-order.

Additionally, an order might visit both the grid cells and the enclosing blocks, for instance, ordering both the blocks in Figure 2.6a and the constituent grid cells in Figure 2.6b. To accomplish this order, one convention is to visit enclosing regions (the blocks in Figure 2.6a) before visiting smaller regions (the cells in Figure 2.6b), as shown in Figure 2.6c, which also includes the top level cell (the enclosing space) in the ordering. In essence, this a hierarchical traversal of the nodes, as shown in Figure 2.6d, which is a preorder tree traversal.

To traverse points in a linear order, the grid cells can be made small enough such that each point is in its own grid cell. To traverse objects in a linear order, either a point in the objects, such as the centroid, is used to represent each object or the objects are assigned to the smallest enclosing block or grid cell, which is similar to creating an MBR for the object, but with more dead space, as shown in Figure 2.7a. Note that an object, no matter how small, that intersects the center point will always be in the top level cell (root space), as shown in Figure 2.7b. Some



(a) (b) (c)

(d)

Figure 2.6: (a) A four cell grid traversed in a Z-order. (b) A sixteen cell grid traversed in a Z-order. (c) A sixteen cell grid traversal that includes enclosing blocks (lettered blocks). (d) A sixteen cell grid traversal as a tree traversal.

algorithms can take advantage of the regular structure of the enclosing cells, but at the price of more false hits due to the increased dead space (see Sections 2.2.4 and 2.3.6).

2.2 The Filtering Stage – Internal Memory

During the filtering stage, a spatial join is performed on approximations of the objects. This section describes techniques for performing a spatial join without using external memory, that is, no data is written to external memory (only read once if necessary). In particular, we note that if there is insufficient internal memory to process a spatial join entirely in memory, then external memory must be used to

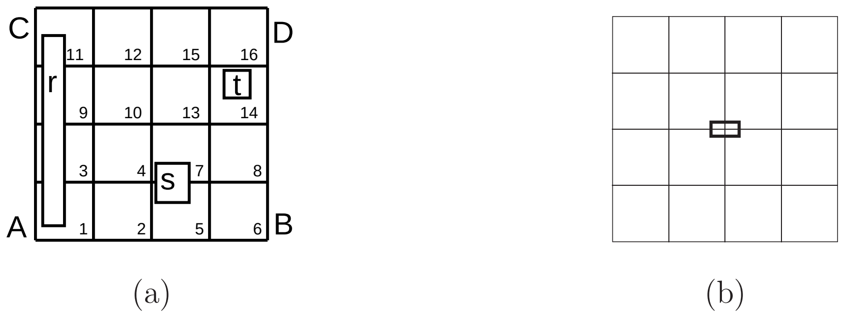


Figure 2.7: (a) In a linear ordering, objects can be assigned to the smallest enclosing block or grid cell. Object r is assigned to the root space since it is not within any block. Object s is assigned to the lower right block, B, and object t is assigned to cell 14. (b) An object overlapping the center point, no matter how small, will be assigned to the top level, which is the entire space.

store all or portions of the datasets during processing (see Section 2.3). Even so, at some point, most external memory spatial join algorithms reduce the size of the problem and process subsets of the data using internal memory techniques.

Section 2.2.1 first describes the brute force nested-loop join. Next, Section 2.2.2 describes the related index nested-loop join, which is presented as an internal memory method even though it can be used as an external memory algorithm if the indices are stored in external memory. Two more sophisticated approaches are described here: the plane-sweep algorithm, rooted in computational geometry, in Section 2.2.3, and a variant of the plane-sweep that uses a linear ordering of the data, in Section 2.2.4.

```
procedure NESTED_LOOP_JOIN(setA, setB, joinCondition)
begin
  foreach a ∈ setA do
    foreach b ∈ SetB do
      if SATISFIED(a, b, joinCondition) then
         REPORT(a, b);
      endif;
    enddo;
  enddo;
end
```

Figure 2.8: The basic nested-loop join with running time $O(n_a \cdot n_b)$, for datasets of size n_a and n_b, using an arbitrary join condition (`joinCondition`).

2.2.1 Nested-Loop Joins

The most basic spatial join method is the nested-loop join, which compares every object in one dataset to every object in the other dataset [136]. The algorithm, shown in Figure 2.8, takes every possible pair of objects and passes it to the `SATISFY` function to check if the pair of objects meets the given join condition, termed `joinCondition` in the algorithm. If a pair satisfies the join condition, then it is reported using the `REPORT` function. Given two datasets, A and B, with n_a and n_b objects in each, respectively, the nested-loop join takes $O(n_a \cdot n_b)$ time. Despite this larger cost, the nested-loop join can be useful when there are too few objects to justify the overhead of more complex methods. Note that this algorithm works with any object type and with any arbitrary join condition.

2.2.2 Indexed Nested-Loop Join

A variant of the nested-loop join algorithm, called the indexed nested-loop join [56], improves performance for larger datasets by first creating a spatial index on one dataset, say A. In this algorithm, given in Figure 2.9, the spatial index is first created and every element of dataset A is inserted into the index using the `INSERT` function. Next, the other dataset, say B, is scanned, and each element is used to search the index on dataset A for intersections. The index is searched using the `SEARCH` function, which in this context becomes a *window query* [68] on the index, where the window is the object from dataset B. Generally, the search window is a rectangle, which limits the types of join conditions to intersection tests or related relations that can be solved with a window query, such as proximity. Typically, for each object in dataset B, the time to search the index is, on average, $O(log(n_a)+f)$, where n_a is the size of dataset A and f is the number of intersections found. For example, the search and insert time, on average, for an R-tree [76] is $O(log(n_a) + f)$. In theory, an object could intersect every object in the index, creating an $O(n)$ search time. In practice though, the number of intersections is small and the running time of the entire algorithm is $O((n_a + n_b) \cdot log(n_a) + f)$, which includes the time to construct the index, which is typically $O(n_a \cdot log(n_a))$. Since all of dataset A is inserted first, more efficient static indices and bulk-loading techniques [17, 84, 104, 191] can be used to improve the construction time and the performance of the index.

The indexed nested-loop algorithm can be executed entirely in memory, using

```
procedure INDEX_NESTED_LOOP_JOIN(setA, setB)
begin
  spatialIndex←CREATE_SPATIAL_INDEX(setA);
  foreach a ∈ setA do
    spatialIndex.INSERT(a)
  enddo;
  foreach b ∈ setB do
    searchResults←spatialIndex.SEARCH(b)
    REPORT(searchResults)
  enddo;
end
```

Figure 2.9: An indexed nested-loop join improves the performance of the spatial join to $O((n_a + n_b) \cdot log(n_a) + f)$, assuming search times of the index are $O(log(n_a) + f)$, where f is the number of results found, n_a is the size of the indexed dataset, and n_b is the size of the unindexed dataset.

in memory indices, and is useful as a component in other spatial join algorithms (see Section 2.3). The algorithm can also be used as a stand alone external memory spatial join algorithm by using external memory indices, which allows the algorithm to process larger datasets. For instance, Becker et al. [24] used grid files [140] as the index and Henrich and Möller [80] used an LSD tree [81] as the index. However, more sophisticated methods exist for using an external spatial index to perform a spatial join (see Section 2.3).

2.2.3 Plane Sweep

A two-dimensional plane-sweep [160] of a set of axis-aligned rectangles finds all of the rectangles that intersect. The algorithm has two passes. The first pass sorts the rectangles in ascending order on the basis of their left sides (i.e., x coordinate values) and forms a list. The second pass sweeps a vertical scan line through the sorted list from left to right, halting at each one of these points, say p. At any instant, all rectangles that intersect the scan line are considered *active* and are the only ones whose intersection needs to be checked with the rectangle associated with p. This means that each time the sweep line halts, a rectangle becomes active, causing it to be inserted into the set of active rectangles, and any rectangles entirely to the left of the scan line are removed from the set of active rectangles [6]. Thus, the key to the algorithm is its ability to keep track of the active rectangles (actually, just their vertical sides), as well as performing the actual intersection test.

To keep track of the active rectangles, the plane-sweep algorithm uses a structure (referred to as a *sweep structure* or `sweepStructure` in Figure 2.10) that supports three operations needed to track the active rectangles. The first, `INSERT`, inserts a rectangle by adding it to the active set. The second, referred to as `REMOVE_-INACTIVE`, removes from the active set all rectangles that do not overlap a given rectangle (or line). These rectangles become inactive when the sweep line halts. The third operation, `SEARCH`, searches for all active rectangles that intersect a given rectangle and outputs them. Examples of structures that support these operations are discussed later in this section.

The classical rectangle intersection problem, given a set of rectangles, S, determines the pairs of intersecting rectangles in S. A spatial join, given two sets of rectangles, A and B, determines all pairs of intersecting rectangles in A and B — that is, for each rectangle r in A, find all of the rectangles in B intersected by r.

[6]A variant of the plane-sweep algorithm also stops at the right sides of each rectangle, which also must be included in the original sorted list, and removes that rectangle from the active set.

```
procedure PLANE_SWEEP(setA, setB)
begin
  listA←SORT_BY_LEFT_SIDE(setA);
  listB←SORT_BY_LEFT_SIDE(setB);
  sweepStructureA←CREATE_SWEEP_STRUCTURE();
  sweepStructureB←CREATE_SWEEP_STRUCTURE();
  while NOT listA.END() OR NOT listB.END() do
    /* get leftmost rectangle from the two lists */
    if listA.FIRST() < listB.FIRST() then
      sweepStructureA.INSERT(listA.FIRST());
      sweepStructureB.REMOVE_INACTIVE(listA.FIRST());
      sweepStructureB.SEARCH(listA.FIRST());
      listA.NEXT();
    else
      sweepStructureB.INSERT(listB.FIRST());
      sweepStructureA.REMOVE_INACTIVE(listB.FIRST());
      sweepStructureA.SEARCH(listB.FIRST());
      listB.NEXT();
    endif;
  enddo;
end
```

Figure 2.10: A two set plane-sweep algorithm to find the intersections between two sets of rectangles.

To apply the plane-sweep algorithm, a sweep structure is needed for both A and B. Rectangles from A are inserted into A's sweep structure and rectangles from B are inserted into B's sweep structure. Also, a rectangle r from A will perform a search on B's sweep structure, thereby finding all the intersections with the rectangles in B, and vice versa. Such an algorithm in given in Figure 2.10.

The data structure used to implement the sweep structures in Figure 2.10 can have a significant impact on performance, as Arge et al. [19] show in their performance studies. The choice of a simple list structure or a block list structure (multiple objects in each list entry) [19] is appropriate for smaller datasets, where the overhead of more sophisticated structures is not needed [50]. For larger datasets or highly skewed datasets, more sophisticated structures are appropriate. Some examples of data structures that will work as sweep structures are:

1. A simple linked list [46].
2. Interval tries [113], as used by Dittrich and Seeger [50].
3. A dynamic segment tree [46].
4. An interval tree [55] with a skip list [161], as described by Hanson [77] and used by Arge et al. [19].
5. A dynamic priority search tree [134].

Except for the linked list implementation, the search operation on the sweep structure is $O(log(n))$, where n is the size of the combined datasets $(n_a + n_b)$, giving a running time for the plane-sweep algorithm of $O(n \cdot log(n))$, which includes the initial sort of the data.

Arge et al. [19] modify the plane-sweep algorithm slightly to dramatically increase the size of datasets that can be processed without resorting to external memory. The traditional version of the plane-sweep algorithm assumes that all of the data is in internal memory. If the data is in external memory, then the entire dataset is first read into internal memory before performing the plane sweep. Arge et al. [19], revisiting work by Güting and Schilling [75], observed that only the data in the sweep structures needs to be kept in internal memory. If the data is in external memory and sorted, then each object can be read one at time from external memory, inserted into the sweep structure, and then purged from internal memory when it is deleted from the sweep structure. In this way, only the data intersecting the sweep line needs to be kept in internal memory, reducing the internal memory requirements of the algorithm and increasing the size of datasets that can be processed without resorting to more sophisticated spatial join techniques. A rough calculation estimates that a typical dataset will have $O(\sqrt{n})$ objects intersecting the sweep line [147], meaning that datasets of size $O(m^2)$, can be processed, where m is the number of objects that can fit in internal memory. The plane-sweep technique can also be extended to process datasets of any size by using external memory (see Chapter 3).

2.2.4 Z-Order Methods

The plane-sweep method described in Section 2.2.3 only uses input sorted in one dimension, but can be adapted to use more than one dimension using a more general linear ordering that sorts the points using multiple dimensions (see Section 2.1.3). Thus, instead of a sweep line, a point or grid cell is swept over the data space, creating an *active border* [14, 49]. Since fewer objects will intersect a point than will intersect a line, the sweep structure will be kept smaller, decreasing search times and the amount of internal memory needed. However, since the enclosing cells used for a linear ordering are bigger than MBRs, the number of objects in the sweep structure will increase, thereby offsetting some of the benefit. Orenstein [143] first used a variation of the Z-order method in his work on spatial joins. This section shows how to adapt the plane-sweep method to use a Z-order (a Peano-Hilbert order would work as well) and relates the algorithm to Orenstein's work.

The Z-order algorithm is nearly identical to the plane-sweep algorithm, shown in Figure 2.10, and this section only describes the two minor modifications needed for the Z-order algorithm, rather than presenting the entire algorithm. First, the objects from both datasets are assigned to Z-order grid cells (see Section 2.1.3). Next, the objects are sorted in Z-order rather than one-dimensionally. The remainder of the Z-order algorithm, which consists of creating the sweep structures and the `while` loop, is identical to the plane-sweep algorithm, shown in Figure 2.10. In this case, the active set, instead of being the objects that intersect the sweep line, are the enclosing cells of the objects that intersect the current Z-order grid cell. Since a sufficiently fine grid cell will intersect fewer objects, the active set will be smaller and the sweep structure can be simpler, such as a linked list [46]. Also, the sweep structure can be modified to take advantage of the regular decomposition of the Z-order cells. All of the objects in the active set (sweep structure), which are the enclosing Z-order grid cells, will have either a containment relation to each other or be identical. In early work on spatial joins, Orenstein [146] used a stack which he called a *nest* to implement the sweep structure. Because the input is sorted in Z-order, large objects can be inserted into the sweep structure before the smaller objects that are enclosed by the object. These small objects will be removed before their enclosing objects are removed. This LIFO property makes a stack the natural choice for the sweep structure. The `INSERT` and `REMOVE_INACTIVE` methods will be simple because they either push elements on to the stack or pop elements from the stack, respectively. The `SEARCH` method is also simple since all objects in the stack will intersect the input object. If the enclosing cells intersect, the MBRs can also be checked for intersection to further filter false hits from the candidate set, assuming the MBRs are available.

Aref and Samet [15] further improved the use of the sweep structure by avoiding some insertions, using a technique similar to a zig-zag join [69]. They point out that if one stack is empty, for example, `sweepStructureB`, then it might not be necessary to insert elements into the other stack, `sweepStructureA`. The objects to skip can be determined by examining the next object to be inserted into the empty stack. For example, in Figure 2.11, if dataset A contains the objects $a1$, $a2$, $a3$ and $a4$, then objects $a2$ and $a4$ will not intersect any objects in dataset B, which contains only $b1$. The objects $a2$ and $a4$ do not need to be inserted into A's stack. If one stack, `sweepStructureB`, is empty, then the algorithm can look ahead to the next element, say $bTop$, that will be inserted into the empty stack, and avoid inserting any elements into the other stack, `sweepStructureA`, that do not intersect $bTop$. In Figure 2.11, `sweepStructureB` will be empty until $b1$ is encountered, which becomes $bTop$. Therefore, $a2$ and $a4$ do not need to be inserted into the stack for dataset A, `sweepStructureA`. In a further extension, Aref and Samet [16] modify the Z-order sweep to report larger pairs first, at each stopping point (iteration of the `while` loop), by reporting from the bottom of the stack up, rather than from the top. Thus, the output is already in Z-order, which can be useful for performing a cascaded join.

One drawback of the Z-order sweep method is that the stacks can be filled with objects that have large enclosing cells, even when the objects are small. For instance, as was shown in Figure 2.7b, any object that overlaps the center point of

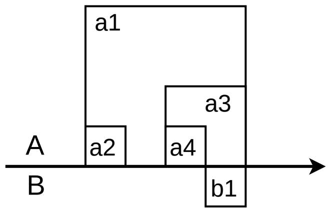


Figure 2.11: Objects are shown from dataset A and dataset B in the order in which they appear in the Z-order, where large objects are encountered first (that is, in the following order: $a1$, $a2$, $a3$, $a4$, $b1$). Since the stack for the B dataset will be empty until $b1$ is inserted, $a2$ and $a4$ do not need to be inserted into A's stack. In other words, when $b1$ is inserted into B's stack, $a1$ and $a3$ are the only elements in A's stack, and $a2$ and $a4$ will never be in A's stack when an element of B is in B's stack. Therefore, $a2$ and $a4$ do not need to be inserted into A's stack.

the space will be contained in the highest level enclosing cell, which encloses the entire space. Such an object will be one of the first objects to enter a stack and will remain in the stack until the algorithm is through processing. This increased stack size will impair performance. To alleviate this problem, Orenstein [144] suggested decomposing objects into multiple cells, as shown in Figure 2.12. This decomposition not only reduces the size of the stack, but creates a more accurate approximation of the object, which reduces the number of false hits. However, these benefits are offset by the increased number of objects introduced by the redundancy and the need to remove duplicate results (see Section 2.1.2). Even so, Orenstein [144] found that performance rapidly improves with a modest amount of decomposition. In a further study, Gaede [67] developed a formula for determining the optimal amount of redundancy.

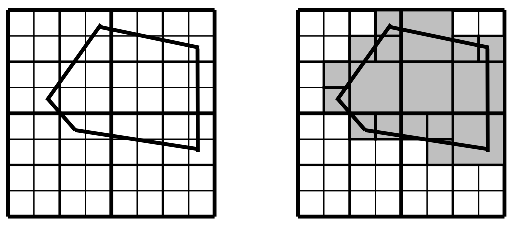


Figure 2.12: An object can be decomposed into multiple cells.

2.3 The Filtering Stage – External Memory

The internal memory techniques described in Section 2.2 require sufficient levels of internal memory in order to operate efficiently. For instance, to perform the nested-loop join (Section 2.2.1), both datasets need to be in internal memory in order to avoid repeatedly reading the same objects in and out of external memory. To efficiently process datasets of any size, an algorithm must use external memory to store subsets of the data (or references to the data) during processing or the data must be indexed. This section describes filtering methods that use external memory to efficiently process datasets of any size.

If the data is already indexed, then it is generally advantageous to use the index for the filtering stage of a spatial join. Section 2.3.1 describes techniques for filtering when both datasets are indexed. Even if there is sufficient internal memory to use the internal memory techniques from Section 2.2, if both datasets are indexed, then it can be faster to use the two-index filtering techniques [7]. Section 2.3.2 addresses the

[7] When an external memory filtering technique should be used instead of an internal memory algorithm is an open question.

case where only one of the datasets is indexed. Of course, the unindexed dataset can be indexed and the two-index techniques from Section 2.3.1 can be used. Another approach, if only one dataset is indexed, is to consider the index as a source of sorted or partitioned data and use the techniques for performing a spatial join when neither dataset is indexed, which are addressed separately, in Section 2.3.3. If neither dataset is indexed, then it might not be efficient to build indices in order to do a spatial join, especially if the indices will not be used again and immediately discarded, as is the case if the spatial join is an intermediate step in solving a complex query.

Even though the internal memory techniques in Section 2.2 cannot be used directly, at some point during processing, two subsets of the data that do fit in internal memory are joined. These subsets can be two pages from indices, as in Section 2.3.1.1, or subsets created by partitioning the data, as in Section 2.3.3. In these cases, when the internal memory techniques from Section 2.2 become applicable, the reader is referred to that section, rather than elaborating on the in-memory join aspects of the particular algorithm.

2.3.1 Both Datasets Indexed

If both datasets are indexed, but with incompatible types of indices, such as an R-tree [76] and a point quadtree [64], Corral et al. [47] suggest ignoring one index and performing an indexed nested-loop join, as described in Section 2.2.2. If both datasets are indexed using the same type of index, then the technique for performing the filtering stage of the spatial join depends on the structure of the index. Since many spatial indices are hierarchical, a spatial join algorithm for these indices also has a hierarchical nature. These methods are described in Section 2.3.1.1. Early work on spatial joins used a more general non-hierarchical approach, which are described in Section 2.3.1.2. Section 2.3.1.3 discusses a method that works with indices that transform objects into points in higher-dimensional space. Since most methods in this section join data pages or index nodes, this issue is discussed separately, in Section 2.3.1.4.

2.3.1.1 Hierarchical Traversal

A common type of spatial index is one that can be described as a *hierarchical containment index* or a *generalization tree* [72, 79], such as an R-tree [76] or a multi-level grid file [203]. This type of index is a tree structure in which every node of the tree corresponds to a region of the data space. An internal node's region covers the regions of its sub-nodes and each node might or might not overlap other nodes, depending on the index type. Each node is typically stored on one page of external memory, which is determined by the underlying DBMS and typically has a size ranging between 1KB and 8KB. This section assumes that the data objects are only stored in the leaves of the tree, though the techniques can be adapted to handle data in the internal nodes. If both datasets are indexed using generalization trees, then a spatial join can be performed efficiently with a *synchronized traversal* of the indices. This section describes a generic synchronized traversal algorithm, and then presents variations that differ in how the indices are traversed, attributable to Günther [72], Brinkhoff et al. [38], Kim et al. [109], and Huang et al. [91] [8].

The generic synchronized traversal algorithm is shown in Figure 2.13. To simplify the explanation, both indices are required to have the same height. For indices of different heights, the join of a leaf of one index with a sub-tree of the other can be accomplished using a window query [38], or handling leaf-to-node comparison as special cases [109]. Starting with the two root nodes of the indices, `rootA` and `rootB`, the algorithm finds intersections between the sub nodes of `rootA` and `rootB` using the `FIND_INTERSECTING_PAIRS` function. The intersecting sub-node pairs are added to the priority queue [46], `priorityQueue`, and these pairs are checked for intersecting sub-nodes in later iterations. If the two nodes are leaves, then the `REPORT_INTERSECTIONS` function is used to compare the leaves and report any intersecting objects. Section 2.3.1.4 discusses the methods used to find object or sub-node intersections within two nodes, as used by the `FIND_INTERSECTING_PAIRS` and `REPORT_INTERSECTIONS` functions.

The three variations of the synchronized traversal algorithm differ in the implementation of the `ADD_PAIR` function, or, in other words, the priority (i.e., the sort order) that is used by the priority queue. Each variation attempts to minimize disk accesses. However, the algorithms must use heuristic methods, as a similar disk scheduling problem for relational joins has been shown to be NP-hard [66, 135]. Günther [72] and Huang et al. [91] both perform a breadth-first traversal, where priority is given to higher-level node pairs. In this way, all of the nodes at one level

[8]See Huang et al. [90] and Theodoridis et al. [187] for cost models for these approaches.

```
procedure INDEX_TRAVERSAL_SPATIAL_JOIN(rootA, rootB)
begin
  priorityQueue←CREATE_PRIORITY_QUEUE();
  priorityQueue.ADD_PAIR(rootA, rootB);
  while NOT priorityQueue.EMPTY() do
    nodePair←priorityQueue.POP();
    rectanglePairs←FIND_INTERSECTING_PAIRS(nodePair);
    foreach p ∈ rectanglePairs do
      if p is a pair of leaves then
         REPORT_INTERSECTIONS(p);
      else
         priorityQueue.ADD_PAIR(p);
      endif;
    enddo;
  enddo;
end
```

Figure 2.13: A generic hierarchical traversal spatial join algorithm for data indexed by hierarchical indices.

of the indices are examined before any nodes in the next level. Since all of the intersecting node pairs for a given level are known before any pair is processed, Huang et al. [91] further sort the pairs for a level, that is, the pairs in the priority queue, to reduce the number of page faults and buffer misses. In this approach, priority is still given to higher level nodes, but a secondary sort is used within each level. They investigated using the following heuristics as a secondary sort:

1. A secondary sort on one dataset's nodes, say A, to achieve clustering for that dataset. For example, for each element of A, say a, all node pairs containing a will be adjacent in the order.

2. A secondary sort on the sum of the centers of the pair in one dimension. In effect, objects pairs whose centers are closer in the $x - dimension$ are given priority.

3. A secondary sort on the center of the MBR enclosing the pair, in one dimension.

4. A secondary sort on the center of the MBR enclosing the pair using a Peano-Hilbert order.

In their experiments, Huang et al. [91] found that ordering by the sum of the centers of the pair in one direction outperformed the other orders in terms of I/O for realistic buffers sizes. One of the drawbacks of the breadth-first approach is that, as the algorithm progresses, the priority queue can grow extremely large and portions of it might need to be kept in external memory.

Brinkhoff et al. [38] and Kim et al. [109] use a depth-first approach in which all of the sub-node pairs for a given node pair, p, are processed before proceeding to the next node pair at the same level. In this case, priority is given to lower-level node pairs. As with the breadth-first approach, heuristics can be used to reduce I/O by secondarily ordering $p's$ intersecting sub-node pairs. Brinkhoff et al. experimented with several ordering heuristics that secondarily sort the intersecting region of the node pairs [9]:

1. In one-dimension.

2. By frequency (maximal degree), determining which node, say a, is contained

[9]A sorted list of intersecting regions of node pairs can easily be determined by using a plane-sweep technique (see Section 2.2.3), which outputs the interesting regions of the node pairs in sorted order. Brinkhoff et al. [38] use a variation of the algorithm described in Section 2.2.3 that does not require a sweep structure, but instead searches the sorted lists of rectangles for intersections.

in the most node-pairs, and processing a's node pairs first.

3. In Z-order.

Brinkhoff et al. found that the Z-order approach worked best for smaller buffer sizes and that the frequency method worked best for larger buffer sizes. To improve performance, Brinkhoff et al. [38] use two *path buffers*, which keep in memory all of the ancestor nodes of the current node pair. They also advocate the use of an LRU buffer to improve performance. Additionally, Brinkhoff and Kriegel [35] suggest that the performance of the spatial join could be improved by organizing the nodes of the index into clusters which are physically close on disk. During join processing, a cluster would be read into memory as a whole.

2.3.1.2 Non-Hierarchical Methods

A more general approach to performing a spatial join on indexed data is to treat the indices as simply a partitioned dataset, where the data pages of the indices are the partitions. In this approach, the data pages are read in an order that is meant to minimize I/O, which is a generalization of the I/O minimization heuristic orders described in Section 2.3.1.1. Each pair of intersecting data pages is then read into memory and joined (see Section 2.3.1.4 for a discussion of joining data pages). This method is applicable to any index type with data pages. Kitsuregawa et al. [110] applied it with k-d trees [28], while Harada et al. [78] applied it with grid files [140].

In this technique [78, 110], the overlapping partitions (data pages) need to be determined first, which is just a spatial join on the areas covered by the data pages,

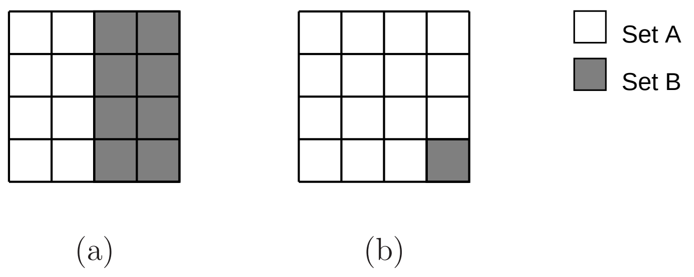


Figure 2.14: When joining datasets A and B, internal memory, represented as data pages in the grid, can be (a) half filled by each of datasets A and B or (b) filled almost entirely with dataset A, leaving only one data page for dataset B.

and internal memory techniques (Section 2.2) can be used to find the overlapping partitions [10]. Once the overlapping partition pairs are determined, partitions are read from one dataset, A, in sorted order [11]. Either enough data pages from dataset A are read to fill half of the available internal memory or enough are read to fill all but one data page of internal memory, as shown in Figure 2.14. Then, the intersecting data pages from the other dataset, B, are read into the remaining internal memory and joined. Since all of the intersecting data pages from dataset B might not fit in memory, the data pages from dataset B are purged from memory once they are joined and more pages from dataset B are read into memory, until all of the intersecting data pages from dataset B have been read. To enhance performance, Lu et al. [125] propose precomputing overlapping index nodes if the index will receive few updates, much like a spatial join index [165].

[10]This approach assumes that the boundary information for the partitions (i.e. the extent of the data pages) is in internal memory.

[11]Note that any linear ordering would suffice (Section 2.1.3).

Corral et al. [47] apply a similar strategy for performing a spatial join between an R-tree [76] (hierarchical and non-disjoint) and a quadtree [64] (hierarchical and disjoint) index. They propose filling internal memory efficiently by reading data pages in groups as shown in Figure 2.14. Additionally, they propose ordering the first dataset of data pages using a linear ordering (see Section 2.1.3). Even though the two indices are of different types, the method works because the data pages of the indices are treated as partitions.

The methods described in this section use heuristics to determine the order of reading partitions. In a relevant analysis, Neyer and Widmayer [139] showed that determining the optimal read schedule to minimize the number of disk reads is *NP-hard* if no restrictions are placed on the partitions. However, if the partitions do not share boundaries (that is, they do not have any sides in common), they show that the problem is easy to solve. If G is a graph where the vertices represent the MBRs of the index nodes and an edge is placed between all of the intersecting nodes between the two datasets, then the optimal read schedule is the Hamiltonian path through G, which can be found using an algorithm by Chiba and Nishizeki [45].

2.3.1.3 Transform to Multi-Dimensional Points

Unlike points, rectangles have extent, which complicates spatial join algorithms. For instance, since an object will not fit neatly into a partition, either the object must be replicated into multiple partitions or the partitions must overlap, as in an R-tree [76]. Transformation methods avoid this problem by transforming objects into multi-dimensional points in a higher dimensional space, such as used by the grid file index [140]. For example, a two-dimensional rectangle can be transformed into a point in four-dimensional space by using the coordinate values of the center point, half of the width, and half of the height as the four values representing the rectangle. Alternatively, the rectangle can be transformed using the coordinate values of the opposing corner points of the rectangle as the four values, which is a technique known as the *corner transformation*.

Song et al. [181] propose a spatial join for data that is indexed using the corner transformation method. The indices create a partitioning of the multi-dimensional points, and the method for joining the partitions is similar to the non-hierarchical spatial join methods (Section 2.3.1.2), which order the processing of overlapping partition pairs between the two datasets. However, in the transformed space, the search space substantially increases. For instance, when joining two indexed datasets, R and S, a region (data page) containing points from a dataset R needs to be compared against a larger region containing points from dataset S. To see why this is so, consider the one-dimensional intervals, a and b, shown in Figure 2.15a. In two-dimensional space, any interval that overlaps a, such as b, will be contained within the region shown in Figure 2.15b, lying between the dashed and dotted lines. Note that all data points in transformed space are above the diagonal since the terminating point of the interval is always greater than the starting point. Similarly, for two-dimensional objects, such as a rectangle r (for example, the MBR of an index node), all rectangles overlapping r will occupy a space in four dimensions similar to the region shown in Figure 2.15b (see Song et al. [181] for the exact calculation

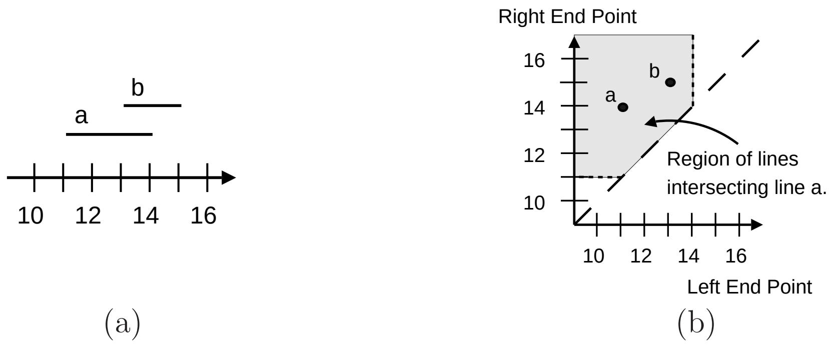


Figure 2.15: As an example of the transformation to multi-dimensional points in a higher-dimensional space, (a) two one-dimensional intervals, a and b, that overlap (b) will be near each other when mapped to two-dimensional points. Any interval that overlaps interval a will be contained within the dotted lines when represented as a point. Furthermore, since the left end point of the interval is represented by the x-axis, all points will be above the dashed, diagonal line since the left end point is always less than or equal to the right end point.

of this region in four dimensions). Once the overlapping partition pairs have been determined, the methods in Section 2.3.1.2 can be used to order the reading of the data pages from memory.

2.3.1.4 Node to Node Comparison

When joining two regions A and B, which could represent two index nodes, two data pages, or two partitions, if both regions cover the same space and fit in internal memory, then every object in region A needs to be joined with every other object in region B. This, of course, is an internal memory spatial join and an appropriate internal memory technique from Section 2.2 should be used. For smaller page sizes, a nested-loop join (Section 2.2.1) might be best because of the low overhead. For larger page sizes, the plane-sweep method (Section 2.2.3), as suggested by Brinkhoff

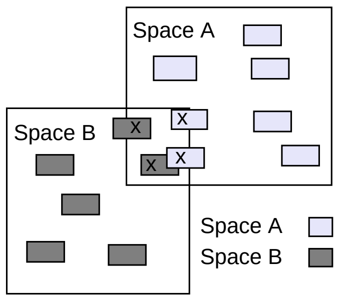


Figure 2.16: When joining two data pages, only the objects within the intersecting region of the pages (marked with an x) need to be considered.

et al. [38], or a Z-order sweep (Section 2.2.4) would be more appropriate.

If the regions do not cover the same space, as is likely when joining index nodes, then the search space can be reduced [38]. Only objects within the intersecting region of the two nodes need to be compared, as shown in Figure 2.16. For example, if the plane sweep method is used to join the nodes, then only these objects will be processed by the plane sweep.

2.3.2 One Dataset Not Indexed

If only one dataset is indexed, then the spatial join can be performed using an indexed nested-loop join (Section 2.2.2), which uses the index to do window queries. For example, given two datasets to be joined, A and B, if dataset A is indexed with an R-tree, then for every element b in B, a window query is performed on the R-tree index of A using each b, which finds all of the objects in A that intersect b. Another approach is to construct an index on the unindexed data and then use the spatial join techniques for when both datasets are indexed, which are described in Section 2.3.1. In support of this approach, techniques for efficiently constructing the second index are surveyed in Section 2.3.2.1. Still another approach is to take advantage of the structure of the indexed data, without using the index directly, as described in Section 2.3.2.2, which reviews techniques that partition the leaves of the index, and Section 2.3.2.3, which reviews methods that adapt the plane-sweep algorithm (Section 2.2.3) to use the index as a source of sorted data.

2.3.2.1 Constructing a Second Index

If only one dataset is indexed, then the other dataset can be indexed efficiently using bulk-loading techniques [84, 191], which exist for many types of indices, and then the techniques described in Section 2.3.1 can be used to perform the spatial join. This approach is especially useful when the index will be saved and used later. Conversely, if the index is not going to be reused, then Lo and Ravishankar [122] suggest building a special purpose index that improves the performance of the spatial join. However, the index might not be reusable because some of the data will be excluded from the specially built index. This constructed index, which is an R-tree [76], is built so that it mirrors the structure of the existing index, thereby minimizing node overlap between the two indices and reducing the number of node-to-node comparisons, which speeds the spatial join.

Lo and Ravishankar [122] call their constructed index a *seeded tree* since it is built using the upper levels of the existing index to seed the construction of the second index. An upper level of the existing index, termed a *seed level*, is used to partition the second dataset. For efficiency, the level should be chosen such that each node is assigned a write buffer. The number of write buffers is limited by the amount of internal memory, which, therefore, determines the lowest level of the index that can be used. Lo and Ravishankar discuss techniques for choosing a level in [123]. In a slightly different approach, Mamoulis and Papadias [129] point out that trying to create too many partitions will cause buffer thrashing and propose a bottom-up approach instead (see Section 2.3.2.3). One level of the existing index forms a collection of non-disjoint regions that do not necessarily cover the data space. This is illustrated in Figure 2.17a and b by regions B and C. Each region's centroid, for example, the center points of the MBRs of the data pages, serves as the basis for partitioning the second dataset. Initially, each partition of the new index has no area. As objects from the unindexed dataset are inserted into a selected partition, the partition is enlarged to enclose all of its objects. Lo and Ravishankar experimented with different techniques for choosing the partition in which to insert objects for the unindexed dataset. The technique that performed the best in their experiments is to insert an object into the partition with the nearest centroid. Another technique, which performed slightly worse, is to insert the object into the partition whose size is enlarged the least.

Once the data is partitioned, each partition is transformed into an R-tree using bulk-loading techniques [191]. The resulting forest of R-trees is attached to the upper seed levels, which are duplicated from the original index, and the new seed levels are adjusted to cover the newly created forest of R-trees, resulting in a regular R-tree. Once this is done, the techniques given in Section 2.3.1.1 can be used to

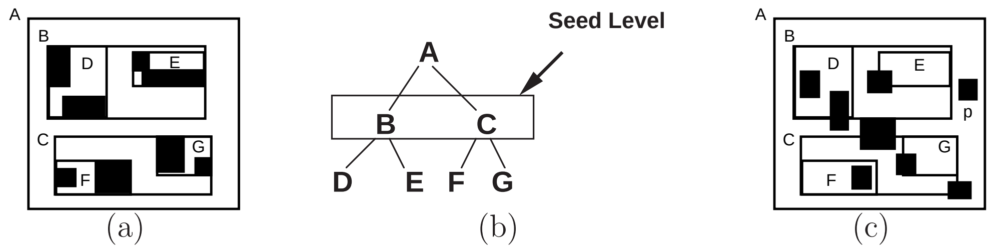


Figure 2.17: (a) The data (dark rectangles) and regions (lettered squares) covered by the nodes of an R-tree. (b) As shown in a hierarchical representation of the index node structure, the second level of the R-tree is used as a seed level. (c) Since object p, from the second, unindexed dataset, does not overlap any seed level region (regions B and C), p can be discarded.

perform the spatial join. Since the trees are similar, performance improves because the number of overlapping nodes is reduced. To improve performance even further, Papadopoulos et al. [152] note that during construction of the R-tree, the leaves of the R-tree (or partitions in general) do not need to be written to external memory to form the full external memory index if the R-tree is not going to be reused. Instead, the leaves or partitions can be joined to the other dataset immediately and then thrown out, saving I/O cost and speeding the spatial join.

Lo and Ravishankar [122] also propose extensions to filter the second dataset, reducing the size of the second dataset and, thus, further speeding the join. Given two datasets A and B, where A is indexed and B is not, they note that any object in B, say b, that does not intersect with the regions covered by the upper level nodes of the existing index on dataset A, could not intersect any object in A. Therefore, b does not need to be inserted into the constructed index for dataset B. This makes the join faster, but renders the second index unusable for processing other operations in the future since it does not contain the entire dataset. For example, if the constructed R-tree is not going to be reused, then objects from the second dataset that do not intersect the regions covered by the seed levels do not need to be inserted into the partitions because they will not intersect any object in the indexed dataset, as shown in Figure 2.17c for object p.

2.3.2.2 An Index as Partitioned Data

Even if just one dataset is indexed, the best approach to performing a spatial join might not be to construct a second index, but rather to use the synchronized traversal methods from Section 2.3.1.1. In this case, the pre-existing index can be viewed as a partition of the dataset and methods similar to the non-hierarchical methods (Section 2.3.1.2) can be used to perform the spatial join. To create partitions from an index, the data pages are grouped to form the partitions. The data pages can be grouped either in a top-down manner for hierarchical indices by using sub-trees as the partitions [192] or in a bottom-up fashion by grouping data pages, which can be used for any index type [132].

In a top-down method to partitioning, proposed by van den Bercken et al. [192], the unindexed dataset is partitioned based on one of the levels of the existing hierarchical index, for example, the sub-nodes of the root of the indexed dataset, which is similar to the seeded tree technique in Section 2.3.2.1 and shown in Figure 2.17b. The partition boundaries are the MBRs of the internal index nodes for the chosen level, for example, regions B and C in Figure 2.17. The unindexed data is partitioned based on these regions, placing each object into each partition that it overlaps, which replicates the data, requiring that duplicate removal techniques be used on the result pairs (see Section 2.3.3.4).

Once the partitioning is done, each partition is joined with the objects in the sub-tree of the corresponding index node using any appropriate internal memory method (Section 2.2). If the data pages and partitions for a sub-node do not fit in memory, then the method can be recursively applied by descending to the next level of the index. This approach works best if the depth of the tree is small or, conversely, if the index has a large fan out. For this reason, van den Bercken et al. [192] propose this approach as a technique for joining two unindexed datasets, in which case an index is created with the largest possible fan out on one dataset before the method is applied.

In a bottom-up approach to partitioning the data pages, Mamoulis and Papadias [129, 132] propose a technique that creates a target number of partitions, which they call *slots*, by grouping the data pages of the existing index, as shown in Figure 2.18. The unindexed dataset is partitioned based on the slots. To create the slots, the amount of data in each slot is first determined, which is roughly half of the available internal memory. The data pages of the existing index are grouped by traversing them in a linear order (Section 2.1.3) and adding them to a slot until the slot's capacity is reached. Then, the next slot is filled and so forth. The region covered by the data pages in the slot form a partition for the unindexed data. As with the top-down approach, once the unindexed dataset is partitioned, a group of data pages from the original index (a slot) and its corresponding partition of the

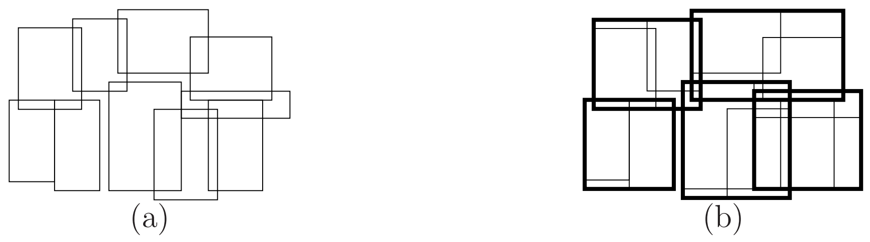


Figure 2.18: (a) A set of index data pages (b) are grouped to form slots, shown by thick-lined rectangles.

unindexed dataset are read into memory and joined. Due to skew, however, a slot and its partition might not fit in internal memory. In this case, the partitioning method is applied recursively until each slot and its partition fit in memory.

2.3.2.3 An Index as Sorted Data

An index can also be viewed as a sorted dataset, since extracting the data from an index in sorted order is inexpensive using an in-order traversal of the index. In this case, the plane-sweep method (Section 2.2.3) can used to perform the spatial join. Generally, the plane-sweep method is performed after sorting both datasets. Since extracting data from an index in sorted order (sorted in one-dimension) is fast in this situation, the plane-sweep technique is less expensive than sorting both datasets since only one dataset needs to be sorted [18].

Another approach, one that modifies the plane-sweep method, proposed by Gurret and Rigaux [74], is to read the data pages from an index (they use an R-tree) in a one-dimensional sorted order and insert entire data pages into the sweep structure. In this case, one sweep structure will contain objects, as is normal, while the other sweep structure will contain data pages. This technique only requires a modification of the plane-sweep `SEARCH` routine (see Figure 2.10) to search for intersections between an object and a data page. Since the `REMOVE_INACTIVE` and `INSERT` routines work with MBRs and the enclosing rectangle of a data page is an MBR, these two methods do not need to be modified. Additionally, the initial sort step of the plane-sweep algorithm needs to extract the data pages of the index as well as to sort the unindexed dataset.

If memory overflows (that is, the active set is too large to fit in internal memory), then Gurret and Rigaux [74] propose using a method in which some of the data pages of the index are removed (or *flushed*) from the sweep structure and written to disk for later processing. To do this, before the plane-sweep phase of the algorithm starts, each data page is assigned to a strip, as shown in Figure 2.19. The strips are created using a method that is similar to forming slots in Section 2.3.2.2, except that a one-dimensional sort is used. Only entire strips of data pages are flushed at a time. After a strip is flushed, the plane-sweep algorithm continues. Any rectangle from the unindexed dataset that overlaps the flushed strip is also written to external memory. After the plane-sweep algorithm finishes, it is run again on the flushed data pages and the rectangles written to external memory, starting from the point where the flush occurred. If a plane sweep on the flushed data also overflows memory, then the flushed data can be partitioned further into strips and the entire algorithm applied recursively.

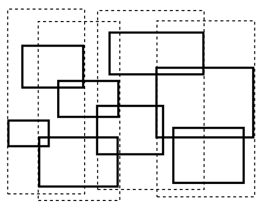


Figure 2.19: The leaves of an R-tree are grouped into strips, which can flushed to external memory if internal memory overflows. Here, four strips (dashed lines) are formed, each containing two objects.

2.3.3 Neither Dataset Indexed

The inputs to a spatial join operation are usually assumed to be indexed. However, the situation is often different when the input to the spatial join are themselves the result of a spatial join. In particular, there is no requirement that the output of a spatial join be indexed (but see Hoel and Samet [88], which evaluates different spatial indices by taking into account the time necessary to construct an index for the result of a spatial join). If neither dataset is indexed, then an index can be built on one or both datasets, and then the techniques from Sections 2.3.1 and 2.3.2, respectively, can be used. If the index is to be saved and reused, this can make sense. If not, then other techniques that do not necessarily create an index might be faster [12]. These techniques are useful when an index would not be used later, such as for a one-time operation on the data, where the extra cost of building the index would be more expensive. For example, in a complex query, intermediate results can be processed faster with these techniques. The key to most of these

[12]Some techniques for unindexed data create a usable index as a byproduct of the spatial join, for example, the filter tree [116].

techniques lies in partitioning the datasets so that the partitions are small enough to fit in internal memory. In other words, a divide-and-conquer approach is used to decompose the datasets into manageable pieces.

Once the data is partitioned, each pair of overlapping partitions, one from each dataset, is read into internal memory and internal memory techniques are used (see Section 2.2). This assumes that the partition pairs fit into internal memory. If they do not, then they can be repartitioned until the pairs fit (see Section 2.3.3.3).

As a foundation for describing the partitioning methods in this section, Section 2.3.3.1 describes a generic algorithm for performing a spatial join using a partitioning technique. The algorithm also serves to introduce several common issues associated with the partitioning approach: determining the number of partitions is discussed in Section 2.3.3.2, repartitioning, if any of the partition pairs do not fit in internal memory, is discussed in Section 2.3.3.3, and handling duplicate results is discussed in Section 2.3.3.4. Next, the various algorithms from the literature are described, grouped by how they partition the data: using grids in Section 2.3.4, using strips in Section 2.3.5, by size in Section 2.3.6, and clustering in Section 2.3.7.

2.3.3.1 Basic Partitioning Algorithm

This section uses a simplified algorithm to illustrate a spatial join technique based on partitioning. While the algorithm has limited practical applications, it is used as the basis for describing more sophisticated algorithms and to discuss the common issues associated with partitioning techniques. The basic concept is to

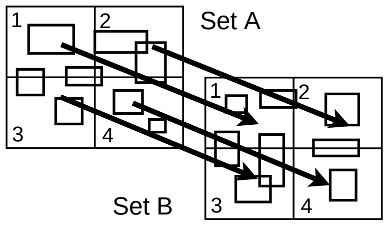


Figure 2.20: If both datasets, A and B, are partitioned using the same grid, then each grid cell is joined with exactly one other, corresponding cell.

define a grid and use it to partition the data from each dataset, datasets A and B, into external memory. Each data object is placed into each grid cell (partition) that it overlaps. Once the data is partitioned, the partitions for each grid cell of dataset A are joined with the corresponding partition of dataset B using one of the internal memory methods from Section 2.2. For example, in Figure 2.20, both datasets A and B are partitioned using the same grid, and then, each corresponding cell is joined with the other. Grid cell 1 from dataset A, for instance, is joined with cell 1 from dataset B, and so forth.

In the algorithm, shown in Figure 2.21, the first step is to determine how many partitions are needed. The goal is to create partitions that are small enough, when paired with a corresponding partition, to fit in internal memory. To do this, the algorithm first calculates the minimum number of partitions needed, `minNumberOf-Partitions`, as the total storage costs of the objects in both datasets divided by the available internal memory. The algorithm uses the `DETERMINE_PARTITIONS` function to create the smallest grid that has at least `minNumberOfPartitions`. However, the calculation of `minNumberOfPartitions` is inaccurate for two reasons. First, if a dataset is skewed, then some grid cells might contain more objects than can fit in internal memory. The more sophisticated techniques described later in this section correspond to different ways of dealing with skewed data. The simplified algorithm assumes that the data is uniform. Second, since a data object is placed into each cell it overlaps, the number of data objects will increase since the object is replicated into each cell. To address this issue, the calculation of the minimum number of partitions can be adjusted using methods described in Section 2.3.3.2. Alternatively, the algorithm can continue and when a partition pair to be joined is encountered that is too large for internal memory, then one or both of the partitions can be repartitioned, creating smaller partitions that can be processed. Section 2.3.3.3 addresses repartitioning. Note that since objects are replicated into multiple cells for both datasets, duplicate results will arise. Section 2.3.3.4 addresses the issue of removing duplicates from the result dataset.

Once the partitions are created, the algorithm scans the datasets, placing each object into each partition that it overlaps using the `PARTITION_DATA` function, which writes the objects to external memory. In the final step of the algorithm, each partition pair is read into internal memory using the `READ_PARTITION` function and joined using the plane-sweep technique from Section 2.2.3 (note that any internal memory spatial join would be appropriate), which will report all of the intersecting pairs between the partitions.

```
procedure GRID_JOIN(setA, setB)
begin
  /* determine the number of partitions */
  m←AVAILABLE_INTERNAL_MEMORY;
  mbrSize←BYTES_TO_STORE_MBR;
  minNumberOfPartitions←(SIZE(setA)+SIZE(setB))*mbrSize/m;
  partitionList←DETERMINE_PARTITIONS(minNumberOfPartitions,
                                    AREA_OF_DATA_SPACE);
  /* partition data to external memory */
  partitionPointersA←PARTITION_DATA(partitionList, setA);
  partitionPointersB←PARTITION_DATA(partitionList, setB);
  /* join partitions */
  foreach partition ∈ partitionList do
    partitionA←READ_PARTITION(partitionPointersA, partition);
    partitionB←READ_PARTITION(partitionPointersB, partition);
    PLANE_SWEEP(partitionA, partitionB);
  enddo;
end
```

Figure 2.21: A simple partitioning technique for performing a spatial join on unindexed data.

2.3.3.2 Determining the Number of Partitions

Many partitioning techniques must first determine a target number of partitions. The goal is to create pairs of partition to be joined that will fit in memory. However, because of data skew, no simple partitioning scheme can guarantee that all of the partition pairs will fit in memory. Therefore, a heuristic calculation must be used. This calculation is constrained by the following factors:

1. The amount of internal memory, which determines the number of objects that can be joined at a time using internal memory techniques.

2. The replication rate, which is the actual number of objects inserted into the partitions, which includes duplicates.

3. The amount of internal memory also limits the number of write buffers that

can be used to partition the data to external memory.

Within these constraints, different techniques either try to maximize or minimize the number of partitions. A minimum number of partitions might be calculated in order to reduce replication, which reduces the number of pairs that need to be joined. A maximum number of partitions minimizes the chances of a costly repartitioning.

The simplest calculation, ignoring data skew and replication, derives the target number of partitions by dividing the total storage costs of the objects by the available internal memory. The total storage costs of the objects is the number of objects multiplied by the size of an object in bytes, referred to as *objectSize*. In an implementation, *objectSize* might be the sum of the size of a rectangle (MBR) in bytes and the size of an object pointer or object key. To account for replication, a scale factor, r, can be added to the calculation. The replication rate depends on the dataset and the partitioning scheme. In one set of experiments [155], the replication rate was found to be 3-10%. Incorporating the replication scale factor, r, the calculation for the minimum number of initial partitions is:

$$\frac{r \cdot (|setA| + |setB|) \cdot objectSize}{m}, \tag{2.1}$$

where m is the available internal memory. However, this equation does not take into account data skew and is only an estimate. In the worst case, with severely skewed data, most of the data could be in one partition. In this case, repartitioning is required (see Section 2.3.3.3).

Equation 2.1 applies to internal memory methods that require all of the data to be read into internal memory before processing begins. Arge et al. [19] showed that the plane-sweep method can process more data than can fit into internal memory, see Section 2.2.3. However, the exact amount of data that can be processed is dependent on the dataset, specifically the *maximum density* of the dataset [61, 98, 187], which is just the maximum number of objects in the active set. However, this value is difficult to calculate for a given dataset.

The maximum number of partitions is limited by internal memory in that internal memory limits the number of external memory write buffers. Typically, for better performance, when writing to external memory, a page of internal memory is filled before it is flushed to external memory. This places a hard limit on the maximum number of partitions.

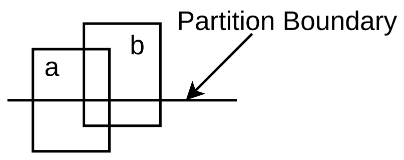
The principal motivation for getting the number of partitions right is to avoid repartitioning (see Section 2.3.3.3). However, the more partitions there are, the greater the replication. Some evidence suggests that this replication does not have a significant impact on the total processing time [213]. In this case, creating the maximum number of partitions might optimize performance. Conversely, internal memory algorithms might perform better with smaller partitions, thereby improving overall performance. From the literature, it is unclear what the best choice is for the number of partitions and thus, we leave this choice as an open question.

2.3.3.3 Repartitioning

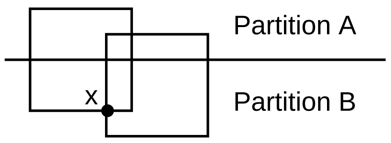
The goal of partitioning is to create partitions that are small enough to be processed by internal memory techniques (Section 2.2). However, the initial partitioning phase (Section 2.3.3.2) might create partitions that are too large because of data replication or skewed data. If this occurs, then the partition pairs that are too large can be further sub-divided using the original partitioning scheme, creating a finer grid. If this repartitioning fails, then the process can be repeated recursively until all of the partition pairs can be processed by internal memory methods.

Repartitioning can occur immediately after the initial partitioning, or it might be necessary to do it later, after some partition pairs have been joined because it might not be known if a repartitioning will be necessary. With the basic internal memory techniques, such as the nested-loop join (Section 2.2), the size of the datasets that can be processed is known, in which case, any over-full partition pairs can be repartitioned immediately, before joining any of the partitions. However, with other internal memory techniques, such as the plane-sweep extension [19], it might not be known whether or not a partition pair is small enough. In this case, repartitioning is done only when an internal memory technique fails because its sweep structure has grown too large for the available internal memory. Any result pairs generated for the current partition pair need to be discarded since the results will be regenerated after repartitioning.

If a partition pair to be joined, say A and B, is too large for the available internal memory, Dittrich and Seeger [50] propose repartitioning only one of the datasets, say A, first, and then joining each sub-partition with B. If this fails to create small enough partitions, then the other dataset, B can be repartitioned. This process can also occur recursively until small enough partitions are achieved.



(a)



(b)

Figure 2.22: (a) An intersection between objects a and b is reported from both partitions into which they are inserted, creating a duplicate result. (b) The reference point method for online duplicate avoidance only reports an intersecting pair if a point in the intersecting region, x, is within the current partition. The point must be chosen consistently, such as always the lower left corner.

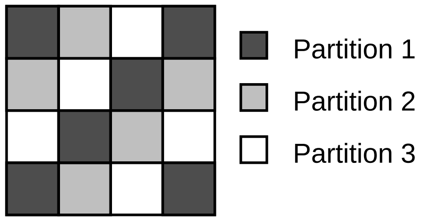
2.3.3.4 Avoiding Duplicate Results

If partitioning the data results in object replication, then duplicate results will be reported. For instance, if a pair of overlapping objects is split by a partition boundary, then the intersecting pair will be reported when each partition is processed, as shown in Figure 2.22a. Some experiments have shown that replication does not add considerably to processing [213], though other experiments have shown the opposite [126]. In either case, duplicate results need to be removed from the candidate set to avoid extra processing during the refinement stage (Section 2.5). The duplicates can be removed with an extra step between the filtering stage and the refinement stage or combined with refinement. With some algorithms, duplicate results can be detected during filtering and removed online.

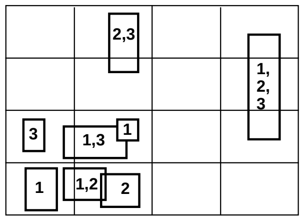
One way to remove duplicate results from the candidate set after the filtering stage is to sort the candidate set and then scan the sorted list, removing duplicates. Some refinement techniques sort the candidate set first, and thus, this duplicate removal technique can be used without a loss of performance (see Section 2.5). However, to sort the candidate set, the entire candidate set must be produced first. In most database systems, that is, demand-driven pipelined systems [71], results need to be produced continuously. In this case, *online duplicate removal* techniques, which remove duplicates as they appear in the filtering stage, are preferred.

To implement online duplicate removal, the internal memory spatial join techniques (Section 2.2) can be modified with a simple test which is applied when the rectangles are checked for intersection. The technique, termed the *reference point method* [14, 50, 174], calculates a point within the intersecting region of the two objects. The pair is reported only if this point is within the current partition. The test point can lie anywhere within the intersecting region of the two objects, such as the centroid of the region or a corner point, but must be chosen consistently [13]. For instance, as shown in Figure 2.22b, reference point x is the lower left corner point of the intersecting region of the two rectangles. Since point x only lies within the B partition, the intersecting rectangles will only be reported when the B partition is processed, but not when the A partition is processed, even though the A partition also includes the rectangle pair.

[13]To use the reference point method, the approximations of the objects cannot be clipped, that is, the full MBR must be stored and not just the portion of the object within a partition (most partitioning methods don't use clipping).



(a)



(b)

Figure 2.23: (a) Noncontiguous grid cells are grouped to form three partitions. (b) Forming partitions from noncontiguous grid cells helps to prevent data skew problems by creating partitions of equal size. Even though the data is clustered in the lower left corner, the partitioning scheme results in each partition containing either four or five objects. Each object is labeled with the partitions in which it will be placed.

2.3.4 Partitioning Using Grids

No technique uses a simple grid as did the generic algorithm in Figure 2.21, since it is only useful for uniform distributions. However, Patel and Dewitt [155] use a uniform grid as a starting point. First, the data space is divided into a uniform grid, where the number of grid cells, which they call *tiles*, is greater (typically much greater) than the number of desired partitions. The grid cells are then grouped into partitions using a mapping function in such a way as to minimize skew by, hopefully, creating partitions that contain a similar numbers of objects. For example, in Figure 2.23a, the grid cells are grouped to form three partitions. Even if the data is skewed, each partition will cover part of the dataset, as shown in Figure 2.23d. Note that the data is not physically partitioned into the grid cells, but only into the final partitions. In other words, grid cells are assigned to partitions first, and then the data is partitioned.

Patel and Dewitt [155] do not proscribe a particular mapping, but only suggest using some form of a hashing function to assign the cells to partitions. The hash function should be chosen to distribute the data evenly amongst the partitions and is dependent on the dataset. As an example, Patel and Dewitt use a round-robin order, that scans the grid in row-major order and alternates assigning the cells among the partitions, as is shown in Figure 2.23a. In the figure, because the grid cells are not contiguous, the partitions have a larger total perimeter, leading to increased data replication since a larger perimeter provides more opportunities for an object to intersect the perimeter. Using more initial grid cells increases the uniformity of the distribution because areas of skewed, dense data are distributed amongst several partitions, but using more grid cells also increases the replication of objects across partitions. Experiments [155] showed that increasing the number of grid cells can create near uniform distributions from skewed datasets, but the replication rate can also increase rapidly. In their experiments, Patel and Dewitt [155] found that Tiger data [190] had a replication rate of about 2.5% with 100 grid cells per partition while Sequoia data [182] had a replication rate of 10%.

Zhou et al. [213] use a variation of this technique in which the data is physically partitioned into the grid cells, first. Since the grid cell sizes are known, the partitions can be formed by grouping contiguous cells until a desired partition size, as determined using methods from Section 2.3.3.2, is reached. This approach reduces the amount of replication because the partition boundary is smaller. The cells can be grouped using any linear order, such as a Z-order (Section 2.1.3). If any grid cells remain after the maximum number of partitions have been formed, then they can be assigned to the partitions with the least objects.

2.3.5 Partitioning with Strips

Arge et al. [19] partition the data into strips so that they can take advantage of their modification to the plane-sweep algorithm (Section 2.2.3). They show a lower bound for their method of $O(n \log_m(n) + t)$ I/O transfers, which is optimal, where n is the number of objects in both sets divided by the block size, B (that is, $n = N/B$, where N is the total number of objects in the datasets), m is the amount of internal memory divided by block size, and t is the number of result pairs divided by block size. For the plane-sweep method, after the data is sorted, it is partitioned into strips that are parallel to the sort direction, as shown in Figure 2.24 [14]. With the modified plane-sweep method, the amount of data that can be processed with a given amount of internal memory is limited by the maximum size of the active set. When the data is partitioned into strips, the maximum size of the active set is the maximum number of objects that intersect the sweep line, which is a function of the height of the strip. By appropriately limiting the height of the strip, which can be a difficult calculation (see Section 2.3.3.2), the width of the strip can be of any length, even infinite, in theory. Also, because strips are used, fewer partitions are needed than in the grid method of Section 2.3.4. Thus, if the grid method needs k partitions to process the data without repartitioning, then the strip method can process the same data with only $\sqrt{k}$ partitions. For instance, if the grid method creates a regular grid with sixteen cells, then the strip method can process the same data with four strips. In other words, the strip method would just ignore the vertical

[14]Güting and Schilling [75] also used a form of strip partitioning to solve the rectangle intersection problem using external memory.

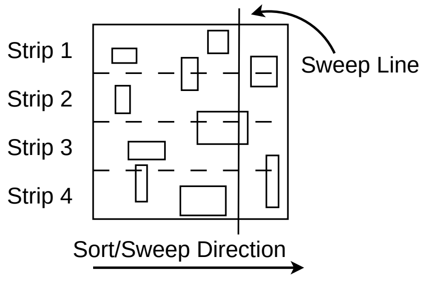


Figure 2.24: In strip partitioning, the data is partitioned into strips and sorted parallel to the strip to take advantage of the plane-sweep modification described in Section 2.2.3. In this example, the sweep is repeated four times, once for each strip. The actual height of the sweep line is the height of the strip.

lines of the grid method. Additionally, since fewer partitions are used, less data is replicated.

2.3.6 Partitioning by Size

Koudas and Sevcik [116] roughly partition the data by size, using a series of finer and finer grids, as shown in Figure 2.25. Each successive grid is derived by subdividing each cell into four equal sized cells. To partition the data, each object is placed into the partition associated with the finest grid in which the object does not intersect the grid lines [15]. For example, as shown in Figure 2.25a, object *a*

[15]In addition to using this structure for the spatial join, Koudas and Sevcik [116] build an index from the level partitions by saving them to data pages and adding an access structure to the data pages. They call the index a *filter tree*, which in essence is an MX-CIF quadtree [106] where an object is associated with its maximum enclosing quadtree block. Arge et al. [19] use a concept similar to the filter tree in conjunction with the strip partitioning method (Section 2.3.5) to avoid inserting large objects into the partitions, improving the performance of the join on each partition

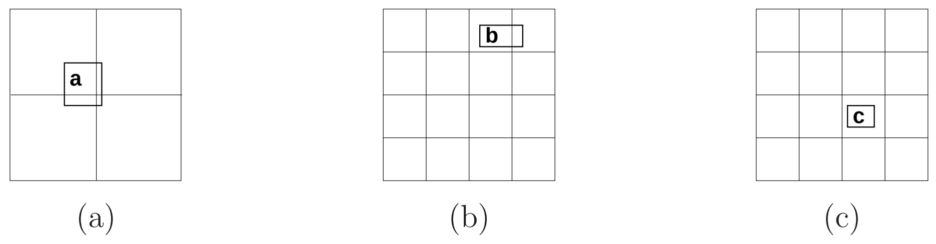


Figure 2.25: To partition by size, objects are associated with the level where they first intersect a grid. (a) Object a is associated with the root space, the highest level, since it intersects the first (coarsest) grid level with four grid cells. (b) Object b is associated with the next level since it intersects the next finer grid level with sixteen cells. (c) Object c is associated with an even lower level.

crosses the coarsest partitioning, and therefore, goes into the first partition. The next partition is similarly composed of objects that do not fit in the first partition and which cross partition boundaries for sixteen equal sized partitions, as shown Figure 2.25b. Each lower level partition, such as the one containing object c in Figure 2.25c, is similarly formed. In essence, each partition is a filter and objects fall through to the lowest level partition where a partition boundary is crossed.

The algorithm for joining two datasets partitioned by size is shown in Figure 2.26. It is a variant of the Z-order method described in Section 2.2.4. The main difference is that the data is partitioned into multiple levels using the `DETERMINE_LEVEL` function and several sweep structures are used for each dataset, one for each level. Note that the data need not be physically partitioned into the levels, but only partitioned into distinct sweep structures, which is a modification suggested by Dittrich and Seeger [50]. The first step in the algorithm is to determine how

and limiting the amount of replication.

many levels are needed, which is done with a simple calculation. The smallest grid cells should be about the same size as the smallest object, which is found using the `FIND_SMALLEST_OBJECT` function. The number of levels then is roughly the base four logarithm of the total number of grid cells at the finest level, which is estimated as the enclosing area of the data space, `TotalArea`, divided by the size of the smallest object, `minSize`. The algorithm then proceeds nearly identically to the Z-order method from Section 2.2.4, accounting for multiple sweep structures. Each object is read (in Z-order) and inserted into its dataset's sweep structure for its level using the `INSERT` function. Next, after the inactive objects are removed using the `REMOVE_INACTIVE` function, the sweep structures for the other dataset are searched for intersections using the `SEARCH` function. Only sweep structures at the same level or shallower (coarser) need to be searched since the Z-order can ensure that larger objects will be seen first. For instance, the two root space sweep structures will be searched after every insertion of the opposing dataset.

The finest level partition needed depends on the size of the smallest objects. The algorithm in Figure 2.26 assumes that all of the coarser levels contain objects, but this might not be the case. The algorithm can be modified to ignore empty levels. Note that because the objects are partitioned by size, they are not replicated between partitions. Also, there is no way to repartition the data. Since the sizes of partitions are determined by grid divisions and not by the number of objects in each partition, there could be too many objects in a particular partition and the sweep structures could overflow the available internal memory. However, in their experiments, Koudas and Sevcik [116] found that sweep structures, which they

```
procedure PARTITION_BY_SIZE(setA, setB)
begin
  /* determine number of levels */
  minSize←FIND_SMALLEST_OBJECT(setA, setB);
  levels←CEILING($\log_4$(TotalArea/minSize));
  listA←SORT_IN_Z-ORDER(setA);
  listB←SORT_IN_Z-ORDER(setB);
  sweepStructuresA[]←CREATE_SWEEP_STRUCTURES(levels);
  sweepStructuresB[]←CREATE_SWEEP_STRUCTURES(levels);
  while NOT listA.END() OR NOT listB.END() do
    /* get next rectangle from the two lists */
    if listA.FIRST() < listB.FIRST() then
      object←listA.POP();
      level←DETERMINE_LEVEL(object)
      sweepStructuresA[level].INSERT(object);
      for( i=level; i<=0; i-- )  do
         sweepStructuresB[i].REMOVE_INACTIVE(object);
         sweepStructuresB[i].SEARCH(object);
      enddo;
      listA.NEXT();
    else
      object←listB.POP()
      level←DETERMINE_LEVEL(object)
      sweepStructuresB[level].INSERT(object);
      for( i=level; i<=0; i-- ) do
         sweepStructuresA[i].REMOVE_INACTIVE(object);
         sweepStructuresA[i].SEARCH(object);
      enddo;
      listB.NEXT();
    endif;
  enddo;
end
```

Figure 2.26: A modified Z-order algorithm that partitions the data by size.

implemented as stacks, tend to contain few objects, and internal memory is unlikely to overflow.

Small objects might be in level partitions meant for larger objects if they cross partition boundaries for that level, as was shown in Figure 2.7b in Section 2.1.3. This hinders performance. Dittrich and Seeger [50] suggest replicating small objects to allow them to be placed into more appropriate partitions. If an object is smaller than the grid cells of the boundary that it overlaps, then the object is divided using the boundary. Each divided piece is then used to find a finer level partition in which to insert the full object. For example, the object in Figure 2.7b would be divided into four pieces and inserted into the four middle partitions. Duplicate removal methods will then be needed (see Section 2.3.3.4).

2.3.7 Data-Centric Partitioning

Lo and Ravishankar [123, 124] cluster the data into partitions. One of the datasets, say dataset A, is first sampled and a nearest-center heuristic is used to identify clusters of the sampled objects. The centroids of the clusters form the basis of the partitions. Initially, each partition is just the identified point, with no area. As objects from dataset A are inserted into the partitions, the partition boundaries are expanded to enclose the inserted objects. Thus, a partition boundary is the enclosing MBR of the objects in the partition. Data is inserted into the partitions using a choice of heuristics, such as inserting into the partition whose size grows the least by area or into the partition whose centroid is closest. Note that this process results in non-disjoint partitions. Once dataset A is inserted into the partition, the resulting partition boundaries are used to partition the second dataset, say B. Objects from dataset B are inserted into each partition that they overlap. Each partition for dataset A then needs to be joined with only one partition from dataset B using any appropriate internal memory method (Section 2.2). Duplicate results do not occur since each object from dataset A is only in one partition. One benefit of building non-disjoint partitions is that the partitions might not cover the entire data space. If this is the case with dataset A, then objects from dataset B that do not overlap any partition can be discarded since they could not be joined with any object in dataset A.

2.4 Filtering Extensions

This section discusses approximations other than MBRs that can be used for filtering, and a method for making the filtering phase non-blocking. Section 2.4.1 surveys techniques for producing better candidate sets with fewer false hits by using better approximations. Section 2.4.2 discusses tests and approximations that can be used to identify true hits, that is, pairs of objects that definitely intersect. These pairs can be immediately reported and then removed from the candidate set, meaning that they are not passed to the more expensive refinement stage. Section 2.4.3 describes a technique for making the filtering phase non-blocking.

2.4.1 False Hit Filtering

The MBR is not the only approximation that can be used during the filtering stage. Other approximations can be used if the filtering method is adapted to use the alternate approximation. Additionally, other approximations can be used as a secondary filter after the initial filtering stage to prune some false hits from the candidate set. To do this, the candidate set is scanned and an intersection test is performed on each pair of objects using a different approximation than was used to do the initial filtering. This second approximation should be stored since calculating the approximation on the fly requires the full object to be read into memory.

Brinkhoff et al. [36] investigated false hit filtering methods that use approximations other than the MBR, which are shown in Figure 2.27. The approximations are:

1. The rotated minimum bounding rectangle.
2. The minimum bounding circle.
3. The minimum bounding ellipse.
4. The convex hull.
5. The minimum bounding n-corner polygon (for example, 5-corner).

They found that using a better approximation generally reduces the number of false hits. The convex hull and 5-corner approximation performed especially well. However, the higher storage costs, the increased complexity of calculating the approximation, and the increased complexity of the intersection test can mitigate the benefit. Even so, Brinkhoff and Kriegel [34] found that additional filtering with these approximations can improve total performance significantly.

Zimbrao and de Souza [218] propose using a 4-color raster approximation for secondary filtering. As shown in Figure 2.28, a grid is imposed over each object and grid cells are colored depending on whether they are covered, mostly covered, partially covered, or not covered. To check for intersections, the two raster approximations of the objects are compared, taking into account the different offsets and

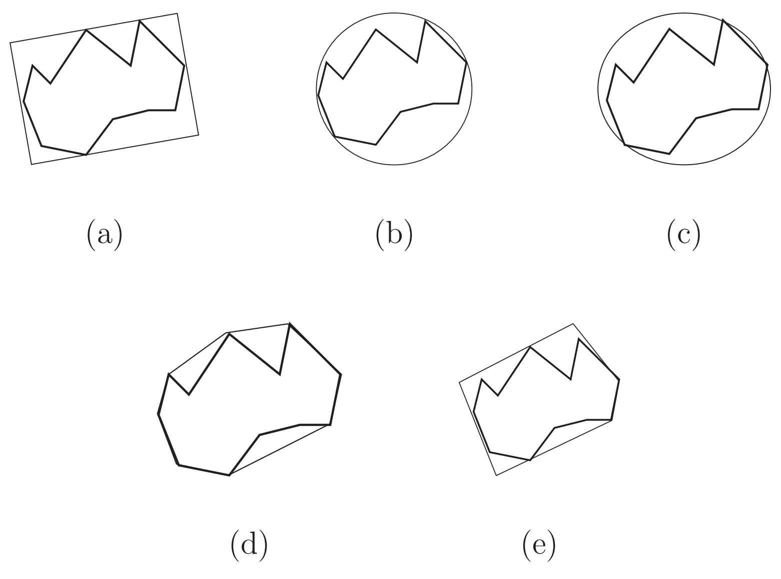


Figure 2.27: Alternate approximations to MBRs include: (a) the rotated minimum bounding rectangle, (b) the minimum bounding circle, (c) the minimum bounding ellipse, (d) the convex hull, and (e) the minimum bounding n-corner polygon (for example, 5-corner).

scales of the grids. The number of grid cells can be adjusted to obtain more accurate results for filtering, but at the expense of higher storage costs.

Veenhoff et al. [198] propose an approximation that is constructed by rotating two parallel lines around the object [16]. In other words, two more sides, parallel to each other, are added by chopping two corners of the MBR, as shown in Figure 2.29, creating a better approximation than the MBR, but still relatively inexpensive to calculate, (i.e., $O(n)$). The tighter approximation reduces the number of false hits.

A number of techniques have been proposed in the spatial indexing, robotics, and computer graphics literature for improving the quality of the approximation

[16]This approximation is also used by Duncan et al. [53] in the BAR tree index.

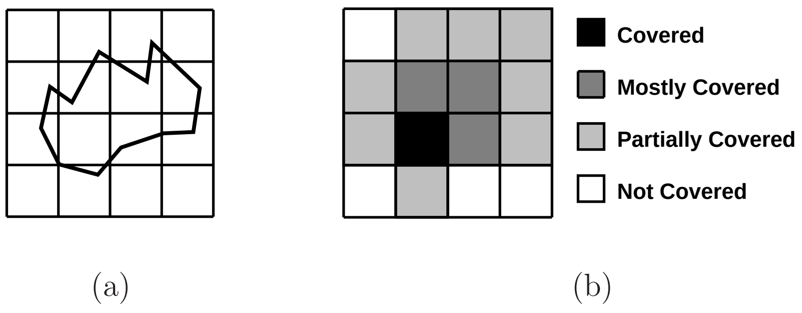


Figure 2.28: An object (a) is approximated using a four color scheme (b) by imposing a grid over the object and coloring the cells depending on whether they are covered, mostly covered, partially covered, or not covered.

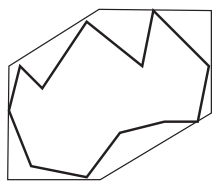


Figure 2.29: An MBR approximation can be improved by adding two more sides, parallel to each other.

yielded by the MBR whose sides are parallel to the axes. For example, an index termed an *oriented bounding box tree* (OBBTree) (for example, [70, 162]) uses a rotated minimum bounding box; a P-tree [100] uses an n-corner polygon; a k-DOP data structure [112], where the number k of possible orientations of the approximation is bounded; and a general spatial filtering mechanism [41], which attempts to find the optimal orientations, uses a minimum bounding polybox. The most general solution is the convex hull, which is often approximated by a minimum bounding polygon of a fixed number of sides having either an arbitrary orientation (for example, the minimum bounding n-corner [52, 173]) or a fixed orientation, usually parallel to the coordinate axes (for example, [59]). The minimum bounding box might also be replaced by a circle, sphere (for example, the sphere tree [93, 141, 194, 204]), ellipse, or intersection of the minimum bounding box with the minimum bounding sphere (for example, the SR-tree [105]). In more than two-dimensions, Kriegel et al. [119] present approximations for high-resolution three-dimensional objects.

In addition to alternate approximations, Koudas and Sevcik [116] propose an additional filtering mechanism that might be useful for sparser datasets. They propose building a bitmap for one of the datasets, say A, where each bit represents a cell in a fine grid over the data space. If any object from dataset A intersects a grid cell, then the bit is turned on for that cell. Then, as objects from the second data are processed, each object is checked against the bit map. If the object does not intersect a cell with the bit turned on, then the object can be discarded.

2.4.2 True Hit Filtering

The traditional filtering phase identifies a candidate set, which is a super set of intersecting objects that also includes pairs of objects that do not intersect, but whose approximations intersect. In true hit filtering, additional approximations are used to find object pairs that definitely intersect, termed *true hits*, and thus can be reported immediately or later combined with the results of the refinement stage. While reducing the size of the candidate set is beneficial, Brinkhoff and Kriegel [34] point out that multiple filters might not combine well, that is, adding more than

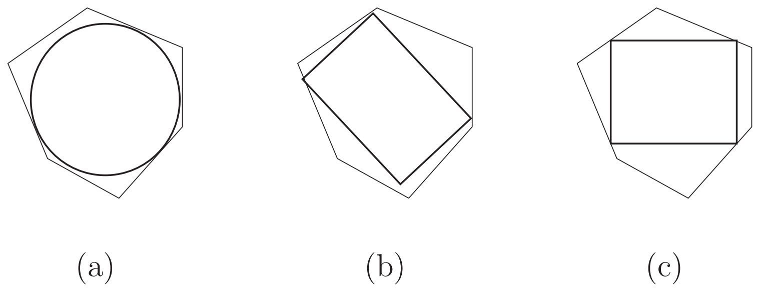


Figure 2.30: An object approximated by: (a) a maximum enclosed circle, (b) a maximum enclosed rectangle, and (c) a maximum enclosed axis-aligned rectangle.

one filter might not significantly improve performance.

To identify true hits, Brinkhoff and Kriegel [34] propose using what they call a *progressive approximation*, where the approximation is entirely enclosed by the full object. In contrast to enclosing approximations, such as the MBR, the progressive approximation does not include extra dead space, but rather excludes portions of the full object. If the progressive approximations of two objects intersect, then the full objects must intersect. Brinkhoff and Kriegel [34] investigated three different progressive approximations, shown in Figure 2.30:

1. A maximum enclosed circle.

2. A maximum enclosed rectangle.

3. A maximum enclosed axis-aligned rectangle.

These approximations were shown to be effective at identifying true hits, but they are more expensive to calculate than MBRs and require extra storage space.

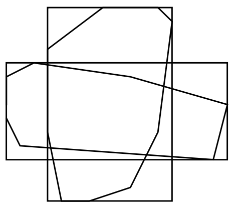


Figure 2.31: If the MBRs of two objects cross, then the two objects must intersect.

Brinkhoff and Kriegel [34] also describe a *cross test* that looks for enclosing approximations, such as MBRs, that cross, as shown in Figure 2.31 [17]. For example, assuming the objects are contiguous, if the MBRs of two objects *c*ross, then the objects must intersect. This is because the objects must also cross at some point and thereby intersect. This is most clearly seen by trying to imagine a counter example where the MBRs cross, as in Figure 2.31, without the objects intersecting. In their experiments, Brinkhoff and Kriegel found that very few intersecting MBRs will cross. However, they argue that since the cross test is so simple, it is worth applying, even if it identifies only a few true hits.

2.4.3 Non-Blocking Filtering

During the filtering phase, any algorithm will block while building an index, sorting the data, or partitioning the data. No output will be produced while this occurs, which delays overall processing in a pipelined system [71]. This section first describes a variant of the indexed nested-loop join (Section 2.2.2) which is non-

[17]The cross test is similar to the intersection test of Ballard [22] and Peucker [158].

blocking and then presents a method to make any filtering technique non-blocking.

The indexed nested-loop join in Section 2.2.2 blocks while it is building an index, that is, it doesn't produce any result pairs, which can slow processing in pipelined systems [71]. To overcome this limitation, Luo et al. [126] use a non-blocking variation of the indexed nested-loop join so that some results are returned immediately. In the algorithm, shown in Figure 2.32, each dataset is indexed using a dynamic index, such as an R-tree [76]. A dynamic index must be used, which can be less efficient than static indices, but static indices block while being built. The input to the algorithm is the combined datasets. If the datasets are separate, as shown in previous algorithms, then either one data element or blocks of elements should be read from each dataset, alternating between the two datasets. Otherwise, no results will be reported until items from the second dataset are encountered. As each element is encountered, it is inserted into the index belonging to it's own dataset and the other index is searched for intersections. In this way, results can appear after only a few objects have been seen, though at the expense of building a second index.

A similar technique can be used to make any filtering technique non-blocking. Luo et al. [126] propose a simple extension, that is conceptually similar to a pipelining hash join [205], and is applicable to most filtering methods. They propose devoting a portion of internal memory to maintaining an in-memory R-tree (or any other spatial index) for each dataset, say datasets A and B, and then using the non-blocking nested-loop join as a front-end to the filtering method. Only a subset of datasets A and B is inserted into the R-trees, which is processed like the non-

```
procedure NON-BLOCKING_NESTED_LOOP_JOIN(combinedDataSets)
begin
  spatialIndexA←INTIALIZE_DYNAMIC_SPATIAL_INDEX();
  spatialIndexB←INTIALIZE_DYNAMIC_SPATIAL_INDEX();
  foreach dataElement ∈ combinedDataSets do
    if dataElement ∈ setA then
       spatialIndexA.INSERT(dataElement)
       intersections←spatialIndexB.SEARCH(dataElement)
    else
       spatialIndexB.INSERT(dataElement)
       intersections←spatialIndexA.SEARCH(dataElement)
    endif;
    REPORT(intersections)
  enddo;
end
```

Figure 2.32: A non-blocking indexed nested-loop join allows results to be reported immediately and continuously.

blocking nested-loop join shown in Figure 2.32. Once the R-trees are full, that is, the allotted internal memory is used up, the remaining data is processed using a more efficient external memory method (Section 2.3). However, the R-trees are still searched using the remaining objects in order to find all of the intersections with the objects in the indices. Once all of the data has been encountered, the objects in the indices can be discarded since all intersections with them have been reported, thereby freeing more internal memory for use by the external memory method used for the remaining objects.

2.5 The Refinement Stage

The filtering stage (Section 2.3) produces a set of candidate object pairs that are typically represented as pairs of object ids. The candidate set contains pairs whose approximations intersect, but do not necessarily intersect themselves. The refinement stage checks the full objects to remove any of these false hits from the candidate set. Unless all of the full objects in the candidate set can fit into internal memory, the key to the refinement stage is ordering the reading of the full objects to minimize I/O. This issue is discussed in Section 2.5.1. Since objects are often polygons, Section 2.5.2 describes a common algorithm for checking if a pair of polygonal objects intersect. This test can be performed faster if the polygons are indexed using a spatial access method [68, 170], which is discussed in Section 2.5.3.

The candidate set might contain duplicate results, which should be removed first to avoid any extra intersection tests on the full objects. If online techniques for duplicate removal are used (Section 2.3.3.4), then the candidate set will not contain any duplicate results. However, some methods cannot use the online techniques and will introduce duplicates into the candidate set. A straight-forward approach to duplicate removal is to sort the id pair list, then scan the list and remove the duplicates. This extra step might not impact performance because it can be combined with techniques for efficiently reading the full objects from external memory in order to do the full object intersection tests, as discussed in Section 2.5.1.

2.5.1 Ordering Pairs

Before performing an intersection test on a pair of objects, each object must be read into memory. Since the full objects might be large, it is unlikely that every object in the candidate set will fit into internal memory. In the best case, each object will be read once, and in the worst case, both objects will need to be read for each candidate pair. The pairs can be processed in the order in which the filtering stage produces them, which is necessary in a pipelined system [71]. Otherwise, if the extra cost of sorting is acceptable, then the candidate pairs can be sorted by ids or by the MBRs using a one-dimensional sort or a linear order (see Section 2.1.3). Sorting improves performance and avoids the worst case of reading two objects for each candidate pair by minimizing the number of repeated reads of an object.

When the candidate pairs are not sorted, Abel et al. [1] showed that the filtering method used impacts the performance of the refinement stage. In their experiments, they showed that the output of Z-order methods (see, for example, Section 2.1.2 and Section 2.3.6) were processed faster in the refinement stage than the output of the hierarchical traversal methods (Section 2.3.1.1). To explain this phenomenon, they suggest that the Z-order methods produce candidates that have more locality, making it more likely that the candidate pairs for an object are near each other in the output order, and thus, the object is more likely to remain in internal memory until it is needed again.

Alternatively, sorting the candidate pairs can reduce the number of times an object is read into memory, although the cost of sorting the pairs might offset some of the performance benefit. The pairs can be sorted using objects from only one dataset or a combination of both objects, using, for example, the centroid of the two objects or the MBR enclosing the two objects. Also, different amounts of internal memory can be devoted to each dataset, which is similar to the techniques used to read partitions in a non-hierarchical spatial join (Section 2.3.1.2). In one approach, the candidate pairs can be sorted for one dataset, say R. The objects in dataset R

will be read into memory only once, while the objects from the other dataset, say S, will be read a multiple number of times. The objects from dataset S might be read as often as once for each candidate set pair, which is still better than the worst case of reading two objects into memory for each candidate pair. In another approach, Patel and Dewitt [155] modify this process slightly by reading as much of the sorted dataset R into memory as possible, leaving room for one object from the dataset S. Then, the objects from dataset S are read one at a time, testing for intersections with each object with which it is paired in the candidate set that is present within the portion of dataset R that is in memory. While objects from dataset R are still read one at a time, objects from dataset S will likely be read less often, since on each read, they can be compared to a multiple number of objects from R.

In a more sophisticated approach for polygonal datasets, Xiao et al. [207] propose clustering the candidate pairs using matrix calculations. In their approach, they pose the read scheduling as an optimization problem that minimizes the number of fetches, weighted by object size. In the matrix, rows represent one dataset and columns represent the other dataset. The matrix values are the sums of the number of vertices in two intersecting polygons or zero for non-candidate pairs. This approach, unlike other methods, takes into account object sizes. Matrix calculations, termed the *Bond Energy Algorithm*, are used to cluster objects into datasets that fit in memory, where an object might be in multiple clusters. This algorithm runs in $O(n^3)$ time, where n is the number of objects.

2.5.2 Polygon Intersection Test

If the objects are polygons, a common approach to test if the objects intersect is a plane-sweep technique [160], which is conceptually similar to the plane-sweep technique described in Section 2.2.3. The algorithm, shown in Figure 2.33, determines if two polygons have any intersecting edges. As with the plane-sweep technique in Section 2.2.3, this algorithm works by sweeping an axis-aligned line across the plane. The end points of the edges from both polygons are the stopping points of the sweep line and the edges intersecting the sweep line form the active set. If the polygons do not have intersecting edges, one polygon could be contained within the other and a separate test for containment needs to be performed, which is described later.

The typical plane-sweep polygon intersection algorithm has been modified to improve its performance for the refinement stage using a technique of Brinkhoff et al. [37]. Since only edges within the intersecting region of the MBRs of the two polygons could possibly intersect, only those edges are used in the plane-sweep method. Any other edges are removed from consideration using the `EDGES_IN_REGION` function in the algorithm. For example, in Figure 2.34, edge e of polygon a is not contained in the intersecting region of the MBRs and could not possibly intersect polygon b.

The algorithm has also been simplified by assuming that the polygons are simple, that is, they do not intersect themselves. In the algorithm, as soon as an intersecting edge is found, the algorithm can report that the polygons intersect

```
function EDGE_INTERSECTION_TEST(edgesA, edgesB, MBRoverlap)
         : boolean
begin
  allEdges←edgesA ∪ edgesB;
  allEdgesInRegion←EDGES_IN_REGION(allEdges, MBRoverlap);
  allEndPoints←DOUBLE_EDGES(allEdgesInRegion);
  edgeList=SORT_BY_END_POINT(allEndPoints);
  activeEdges←CREATE_SWEEP_STRUCTURE();
  while edgeList ≠ ∅ do
    edge←edgeList.POP();
    edgeAbove←activeEdges.EDGE_ABOVE(edge);
    edgeBelow←activeEdges.EDGE_BELOW(edge);
    if edge is the beginning of the edge then
      activeEdges.INSERT(edge);
      if INTERSECT(edge, edgeAbove)
         OR INTERSECT(edge, edgeBelow) then
        return true;
      endif;
    else
      if INTERSECT(edgeAbove, edgeBelow) then
        return true;
      endif;
      activeEdges.REMOVE(edge);
    endif;
  enddo;
  return false;  /* No intersection detected. */
end
```

Figure 2.33: A plane-sweep algorithm for detecting the intersection of simple polygons whose MBRs intersect.

without checking if the intersecting edges belong to the same polygon because the polygons are simple [18].

The first step of the algorithm in Figure 2.33 combines the edges from both polygons into one dataset and removes any edges that do not overlap the intersecting region of the MBRs, `MBRoverlap`, using the `EDGES_IN_REGION` function. Next, the `DOUBLE_EDGES` function creates two entries for each edge, one for each end point,

[18]Becker et al. [23] review and propose algorithms for non-simple polygons and efficient algorithms for finding all of the intersecting edges.

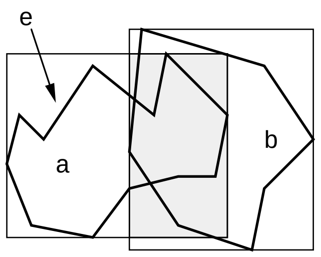


Figure 2.34: Only the edges contained within the intersecting region of the MBRs could intersect each other. Edge e of object a, which is outside of the intersecting region, could not possibly intersect object b.

which are the stopping points of the sweep line. Then, the end points are sorted in one dimension using the `SORT_BY_END_POINT` function. As with the plane-sweep algorithm in Section 2.2.3, a sweep structure is needed to store the edges that intersect the sweep line. In this case, the sweep structure, referred to as `activeEdges`, must perform two additional operations, `EDGE_ABOVE` and `EDGE_BELOW`, which identify edges that intersect the sweep line just above the given edge or just below, respectively. After initializing the `activeEdges` structure with the `CREATE_SWEEP_-STRUCTURE` function, the list of edges is scanned. When an edge is first encountered, it is inserted into the `activeEdges` structure, using the `INSERT` function. When the end of an edge is encountered, it is removed from the `activeEdges` structure using the `REMOVE` function.

In the algorithm, an edge intersection test is performed when an edge is inserted or removed from the sweep structure. As the sweep line moves, the relative positions of the intersection points of the sweep line with the edges change. For example, as shown in Figure 2.35a, the edges intersect the sweep line at position A, in the order 3, 1, 2, and 4, from top to bottom. At position B, the sweep line does not intersect edges 1 or 2 and edge 3 has moved down the sweep line while edge 4 has moved up. The edges continuously change their intersection point with the sweep line as the sweep progresses, but the relative order of the edges only change when an edge is inserted or removed (or when two edges intersect). Furthermore, only the two edges adjacent to the inserted or removed edge will move in the relative order. For instance, when edge 5 is inserted in Figure 2.35b, only edges 2 and 4 have changed order and are no longer adjacent. Therefore, when an edge is inserted, an intersection test is performed with it and the edge above and the edge below using the `EDGE_ABOVE` and `EDGE_BELOW` functions, respectively, to find the edges to be tested. Similarly, when an edge is deleted, only the relative order of the edges above and below it are effected. For instance, when edge 6 is deleted in Figure 2.35c, edges 3 and 4 become adjacent. Therefore, when an edge is deleted, an intersection test is performed between the edge that was above it and the edge that was below it, using the `EDGE_ABOVE` and `EDGE_BELOW` functions, respectively.

Because the polygons are assumed to be simple, if any edges intersect, then the polygons must intersect and true is returned. Otherwise, the sweep completes without finding any intersections and false is returned [19]. Since a polygon could be

[19]In the full polygon intersection test, where all of the intersecting edges need to be found or the polygons are not simple, intersections with adjacent edges also need to be checked when an intersection point occurs since the two edges switch positions in the list, creating new adjacencies for the edges. The algorithm shown in Figure 2.33 just reports success as soon as any intersection is found since the polygons are simple and will not self-intersect.

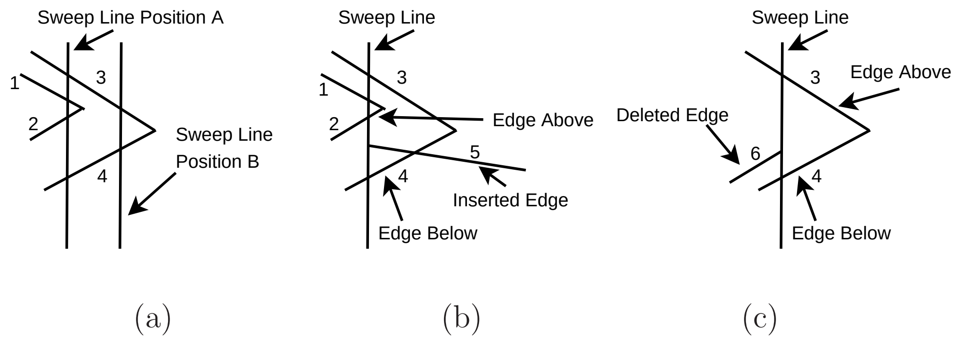


Figure 2.35: (a) Two intersecting edges will intersect the sweep line at adjacent points as the sweep line approaches their intersect point. (b) When an edge is inserted, it could intersect the edge just above it or below it. (c) When an edge is deleted, the edges adjacent to it might intersect.

contained within the other, a containment test is necessary if false is returned to confirm that the polygons do not overlap. A containment test [160], which typically has an $O(n)$ running time, needs to be run twice to conclude that neither object is contained within the other. One such algorithm, briefly described here, takes a point from one polygon, say a, and the edges from the other, say b, and checks if the point is contained within the polygon. If so, then a must be contained within b. Any point on or in polygon a will suffice. Using a horizontal line through the point from polygon a, the algorithm counts the number of edges of b that intersect the horizontal line to the left of the given point. As shown in Figure 2.36, the point is contained within the polygon only if the horizontal line intersects the edges an odd number of times to the left of the point. The algorithm concludes by returning true if an odd number of intersections to the left of the point were detected and

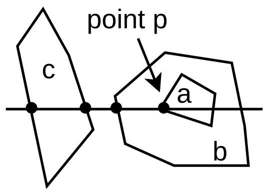


Figure 2.36: Object *a* will be reported as contained within object *b* since the horizontal line through a point, p, in *a* intersects object *b* once to the left of point p. Object *a* will not be reported as contained within object *c* since it intersects the horizontal line twice to the left of point p.

false otherwise. Care must be taken in counting intersections with horizontal edges, which can be counted as either no intersection or as two intersections. Care must also be taken with intersections involving the end points of edges, which can be counted as half of an intersection. For more details on this issue, see the discussion of the point-in-polygon test in any computer graphics book, such as [65].

2.5.3 An Alternate Intersection Test

To improve the speed of the polygon intersection test, at the expense of higher storage costs, a spatial access method can be used to store each full polygon [68, 170]. When the polygons are represented by a containment hierarchy (for example, an R^+-tree [176] or a region quadtree [111]), a synchronized traversal similar to that described in Section 2.3.1.1 can be used to check for intersections between the full polygons. For example, Brinkhoff et al. [37] propose that each full polygon be decomposed into trapezoids, as shown in Figure 2.37, and storing the pieces in an R^*-tree like structure called a TR^*-tree. The entire TR^*-tree for each polygon is stored with the polygon. When two polygons are checked for intersection, both

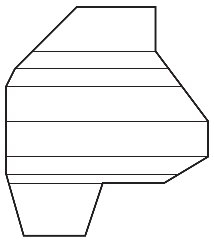


Figure 2.37: A polygon can be approximated by trapezoids, which can be used to perform a faster polygon intersection test.

TR*-trees are read into memory and then a synchronized traversal is performed to check for intersections. This process can be further improved by using heuristics that detect intersections between partial polygons [20, 92].

2.6 Specialized Spatial Joins

A basic spatial join is executed between two sets of objects on the same machine with a single processor. Modifications of the spatial join can be made to improve performance when more than two sets of objects are joined in a *multiway* join, as described in Section 2.6.1. Also, the spatial join requires special considerations when it is performed using a parallel architecture, described in Section 2.6.2, or on distributed computers, described in Section 2.6.3. Section 2.6.4 presents still further variations and approaches to solving spatial joins.

2.6.1 Multiway Joins

A complex query could contain multiple spatial joins. As an example, consider the query – *find all regions between 500 meters and 700 meters above sea level that* *receive 10 to 15 centimeters of rainfall and are in a forested area.* This query requires a spatial join to find the intersecting regions of rainfall and elevation and also the intersection with land type (forest). This query can be answered by performing a spatial join between any two of the datasets and joining the results with the third dataset. The intermediate results are the intersecting regions of the result candidate set, which is then joined with the third dataset. In the example, rainfall and elevation can be joined, creating a new dataset that represents regions with 10 to 15 centimeters of rainfall and which are between 500 and 700 meters above sea level. This intermediate dataset is then joined with the land type data to create the final result.

The previous example illustrates a query with mutual intersection between the datasets. A spatial query might not require that the datasets mutually intersect. For example, the query – *find all roads that cross a river and a railroad track* – does not require that the river and the railroad track in a result triplet intersect. In Figure 2.38a, the three objects have a mutually intersecting region, but in Figure 2.38b, each object pair-wise intersects. Furthermore, it is not necessarily the case that a relationship exists between all of the datasets in the query. For instance, as shown in Figure 2.38c, the query could only require that objects from dataset B intersect an object from both A and C, but no restriction is placed on intersections between A and C, such as with the road, river, and railroad example. In general, a multiway spatial query can be represented by a graph, as shown in Figure 2.39, in which n-ary relations are used to represent mutually intersecting regions. For instance, in Figure 2.39, objects from A, B, and C are required to mutually intersect,

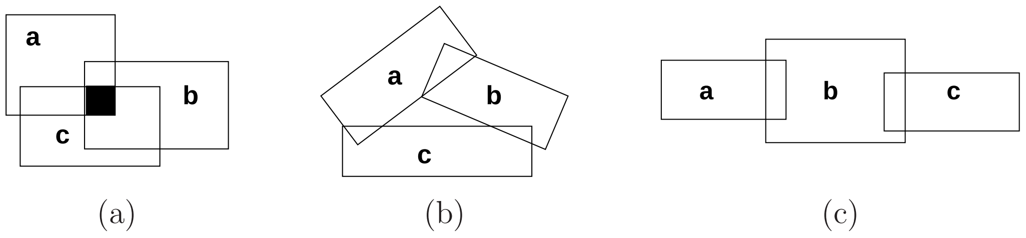


Figure 2.38: Three objects with (a) mutual intersection, (b) pair-wise intersection, (c) two of the objects not intersecting.

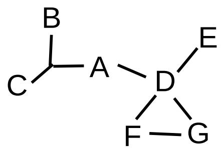


Figure 2.39: A graph representing the relations between datasets in a complex query between multiple datasets. Each edge is a join condition between the objects, which might be more than a binary relation, such as the ternary relation joining A, B, and C.

as in Figure 2.38a, as opposed to objects D, F, and G, which are only required to pair-wise intersect, as in Figure 2.38b.

As mentioned, a multiway spatial join can be solved by joining two datasets at a time and creating intermediate datasets. Furthermore, traditional database optimization approaches [71] that order the pair-wise joins to efficiently solve the query can be used [129]. To do this, selectivity estimates (Section 2.7) are required in order to efficiently determine which pair of datasets to process first, typically processing the pair that produces the smallest intermediate result set first. Moreover, query optimizers also need selectivity information in order to efficiently perform queries that involve a mix of spatial and aspatial data.

To be more efficient, several techniques have been proposed for extending the filtering algorithms (Sections 2.2 and 2.3) to process multiple datasets at once. For example, the nested-loop join (Section 2.2.1) can be extended to process three datasets simultaneously by nesting another loop and using a join condition that takes three arguments, which tests for a three-way intersection. Section 2.6.1.1 describes a similar extension to the indexed nested-loop join (Section 2.2.2), which serves as a basis for describing a more sophisticated technique that is derived from *constraint satisfaction problem* (CSP) techniques [120]. Next, Section 2.6.1.2 describes extending the synchronized traversal spatial join (Section 2.3.1.1) to perform a multiway spatial join.

2.6.1.1 Multiway Indexed Nested-Loop Spatial Joins

A multiway version of the indexed nested-loop algorithm (Section 2.2.2) can be created by nesting each additional dataset. This algorithm, shown in Figure 2.40, is a simplification of algorithms by Mamoulis and Papadias [133] and Papadias et al. [150], and only joins three datasets. More datasets could be joined with further nesting, but the temporary candidate sets can grow significantly larger with more datasets. A generic join condition, termed `joinCondition`, that specifies the desired relation between the datasets, is used since a relation other than intersection might be required. However, each dataset is assumed to be involved in some form of an intersection relation with another dataset. Otherwise, the multiway spatial join might not be appropriate.

```
procedure MULTI_INDEX_NLJ(joinCondition, setA, setB, setC)
begin
  spatialIndexB←CREATE_SPATIAL_INDEX(setB);
  spatialIndexC←CREATE_SPATIAL_INDEX(setC);
  foreach a ∈ setA do
    tempSetB = spatialIndexB.SEARCH(a);
    foreach b ∈ tempSetB do
      tempSetC = spatialIndexC.SEARCH(b);
      foreach c ∈ tempSetC do
        if SATISFIED(a, b, c, joinCondition) then
          REPORT(a, b, c);
        endif;
      enddo;
    enddo;
  enddo;
end
```

Figure 2.40: A multiway indexed nested-loop join that finds objects from three datasets that satisfy any type of multiway intersection (see Figure 2.38).

The first step in the algorithm, shown in Figure 2.40, indexes `setB` and `setC`. Then, for each element of `setA`, a window query is performed on the index of `setB` using the `SEARCH` method. This assumes that the relation specifies that `setA` and `setB` intersect. Each of the search results from `setB` is used to perform a window query on `setC`, which produces another intermediate result set. Note that not all elements of `setC` need to intersect elements of `setA` as intersection is not a transitive relation (for example, objects a and c in Figure 2.38c do not intersect each other, even though they both intersect object b). Finally, each result from `setC`, along with the current value for `setA` and `setB` are checked to ensure that they satisfy the given relation using the `SATISFIED` function, and if so, the values are reported. The `SATISFIED` function ensures that the specified join condition is met. For example, if mutual intersection is required, as in Figure 2.38a, then the intersections of the type shown in Figures 2.38b and c will not be reported. Note that the algorithm explicitly instantiates the temporary result sets, for example, `tempSetB`. This is done in order to give the reader an idea of the internal memory requirements of the algorithm. A database iterator could be used instead, thereby letting the database engine optimize storage and retrieval of the temporary result set.

To enhance the performance of the algorithm, if the query seeks mutually intersecting rectangles, each search can use the intersections of the current values of the previous datasets to further improve performance. For instance, when the index on `setC` is searched, the intersection of `a` and `b` would be the search window, rather than just `b`. Papadias et al. [150] refer to this approach as a *window reduction*. In a hybrid approach, Papadias et al. [150] propose performing a normal spatial join on two of the datasets first, which should be faster than a nested-loop join, and then using window reduction for the remaining values.

To enhance the performance of the algorithm further, each temporary result set could be checked against all previous relations to further prune the result set. For example, the algorithm only considers the relations between `setA`/`setB` and `setB`/`setC`. A relation could exist between `setA` and `setC`, as in Figure 2.38a. If more datasets were being joined with further nesting, then reducing the size of `tempSetC` would be advantageous. In this case, after the temporary result set for `setC` is generated (`tempSetC`), it could be checked against the current value from `setA`, `a`, to further reduce the size of `tempSetC`.

The previous enhancement can be carried even further by having each current set immediately prune each following dataset. For instance, in the algorithm in

Figure 2.40, each value from `setA` could be used to produce temporary sets for every other dataset, `setB` and `setC`, before scanning any other dataset. Papadias et al. [150] and Mamoulis and Papadias [133, 130] use this approach to solve the multiway spatial join, which is a constraint satisfaction problem (CSP) technique [120] called *multi-level forward checking*. In the CSP approach, temporary result sets are generated as soon as possible. In this algorithm, shown in Figure 2.41, if a relation exists between `setA` and `setC`, then a temporary result set, `tempSetC`, is generated in the outer loop using the current value from `setA`. If a relation exists between each dataset, then all temporary results are generated in the outer loop with each object from `setA`. These sets are pruned as each object is instantiated for each dataset in the nested loops. As a further possible enhancement, another CSP technique is to dynamically vary the order in which the datasets are processed, choosing the one with the smallest temporary result set first.

2.6.1.2 Multiway Hierarchical Traversal

Papadias et al. [150] and Mamoulis and Papadias [133, 130] extend the hierarchical traversal method (Section 2.3.1.1) to do a multiway spatial join. This technique applies if each dataset is indexed using a hierarchical index, such as an R-tree [76]. For two datasets, the original method compares overlapping nodes of the two indices. If the nodes are leaves, the intersecting objects within the node are reported, else the intersecting child node pairs are placed on a priority queue to be processed later. To do a multiway join, the queue is modified to hold multi-

```
procedure MULTI_INDEX_CSP(joinCondition, setA, setB, setC)
begin
  tempSetA←∅
  tempSetB←∅
  foreach a ∈ setA do
    foreach b ∈ setB do
     if SATISFIED(a, b, joinCondtion) then
        tempSetB←tempSetB ∪ b
     endif;
    enddo;
    if relation between setA and setC then
       foreach c ∈ setC do
        if SATISFIED(a, c, joinCondtion) then
           tempSetC←tempSetC ∪ c
           endif;
       enddo;
    else
       tempSetC = setC;
    endif;
    /* inner loop */
    foreach b ∈ tempSetB do
       foreach c ∈ tempSetC do
          if SATISFIED(a, b, c, joinCondtion) then
             REPORT(a, b, c);
          endif;
       enddo;
    enddo;
 enddo;
end
```

Figure 2.41: A multiway indexed nested-loop join that finds objects from three datasets that satisfy any type of multiway intersection using constraint satisfaction problem (CSP) techniques to prune temporary datasets.

ple nodes, one from each dataset, instead of pairs. To start, the root nodes of the indices are checked and combinations of the root node's children that satisfy the join condition are placed on the queue. Next, the first set of nodes on the queue is checked and combinations of children are put on the queue. This process repeats until the queue is empty. Papadias et al. [150] and Mamoulis and Papadias [133] suggest using multi-level forward checking (Section 2.6.1.1) to check the nodes, but any appropriate multiway version of an internal memory spatial join could be used. For instance, Park et al. [153] propose using a pair-wise plane-sweep method for comparing multiple nodes at once, that is, a plane-sweep method is used on the first two datasets and the results are joined with the third dataset with the plane-sweep method, and so forth. As an alternate hybrid approach, Papadias et al. [151] propose using the hierarchical traversal approach to instantiate the first few variables, then finishing with window reduction (Section 2.6.1.1) to instantiate the remaining variables.

2.6.2 Parallel

In a parallel architecture, work is distributed amongst several processors. For a spatial join, the work can be distributed in both the filtering and refinement stages, and also for partitioning unindexed data. For the filtering stage, parallel techniques that extend the synchronized hierarchical traversal approach (Section 2.3.1.1) have been used for indexed data [39], and techniques that extend the grid partitioning methods (Section 2.3.4) have been used for unindexed data [126, 156, 213]. These techniques assume a shared nothing architecture (each processor has it's own memory), although some algorithms have extensions that use shared memory architectures to improve performance [39, 213]. Most of these methods have shown a near linear speed increase with more processors. However, the key to achieving linear speed increases during filtering depends on load balancing. An algorithm that uses a specialized hypercube architecture to join two indexed datasets is also discussed

[87].

Brinkhoff et al. [39] investigated extending the synchronized hierarchical traversal approach (Section 2.3.1.1) to parallel architectures by assigning sub-trees of the index to each processor. In the simplest approach to distributing the work amongst n processors, a single processor first performs a hierarchical traversal, one level at a time, until the number of node pairs on the priority queue exceeds n. At that time, the node pairs are distributed to the processors, which perform a sequential spatial join. However, as shown in Figure 2.42, one processor might be assigned nodes $a1$ and $b1$ and another processor the node pair $a1$ and $b2$. In this case, node $a1$ will be read from memory twice. To overcome this inefficiency, Brinkhoff et al. propose using global buffers (in a shared memory architecture) so that one processor can access a node that another processor has read into internal memory. However, this approach does incur extra communication costs to synchronize usage of the buffers. Additionally, the workloads might not be balanced since the processing time for each sub-tree could be different. Rather than assigning all of the node pairs from the queue at once, Brinkhoff et al. suggest assigning only one node pair at a time to each processor. Once a processor finishes with one node pair, another is requested while the queue is not empty. When the queue is empty, they further suggest that one processor can share the work of another processor by taking node pairs from another processor's local queue, i.e., sub-trees of the original sub-trees.

For unindexed data, Zhou et al. [213] adapt the variation of the grid partitioning method (Section 2.3.4) that physically creates the grid cells. In their approach, the random data is evenly divided amongst the n processors, which then partition

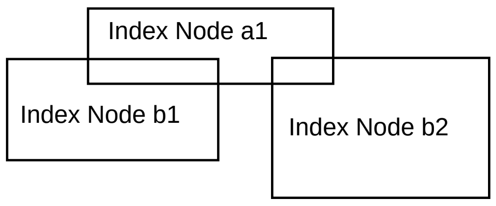


Figure 2.42: Objects in multiple partitions might be read from memory multiple times.

the data using the same tiling scheme. Then, with the sizes of the grid cells known, a single processor determines the merging of grid cells into n partitions using a Z-order merge, creating, on average, an even load distribution amongst the processors. Next, each processor is assigned a partition and the data is redistributed appropriately. Finally, each processor filters the data using a sequential spatial join filtering technique. Both Patel and Dewitt [156] and Luo et al. [126] investigated a similar approach. Rather than physically tiling the data, though, they both use the virtual tiling method (Section 2.3.4), relying on a good hash function to distribute the data evenly.

For parallel refinement, each processor could refine the candidate pairs it produces. However, Zhou et al. [213] point out that it is difficult without much selectivity information to balance the number of candidates produced by each processor during the filtering stage. If the candidate pairs are redistributed, then the workload for refinement can be close to optimal since the number of vertices in each polygon are good estimates of performance. Like sequential refinement, performance can be improved by ordering the candidate set to minimize the number of times each full object is read. Zhou et al. argue that an efficient approach is to use one processor to sort the candidates into a linear order, and then assign the candidates to each processor in a round-robin fashion. Each processor will be processing candidate pairs that are near each other (exhibiting locality), increasing the likelihood that an object only needs to be read into internal memory once (assuming a shared memory architecture), rather than continually reading some of the same objects throughout refinement. Zhou et al. took the approach of assigning full objects to processors, which are responsible for reading the object into memory and distributing the object to other processors. They found that the redistribution costs are negligible.

Hoel and Samet [87] describe parallel algorithms for PMR bucket quadtrees [138] and R-trees [76] using a specialized hypercube architecture. The algorithms require that the data fit in memory, but result in extremely fast algorithms. In their quadtree approach, regions of one of the quadtrees are associated with half of the nodes (processors), termed the *source nodes*, and corresponding regions of the other quadtree are associated with the other nodes, termed the *target nodes.* Each quadtree is assumed to cover the same area, and thus, because of the regular decomposition, there is a one-to-one correspondence between source nodes and target nodes. The source nodes send their objects to the target nodes, which check for intersections. Hoel and Samet also describe a similar algorithm for R-trees in which the index nodes of the R-tree are associated with the processor nodes. However, because of the irregular decomposition, each source node will be associated with multiple target nodes, dramatically increasing the communication costs and slowing the join.

2.6.3 Distributed

In a distributed spatial join, the datasets reside at different locations. Abel et al. [2] show how to combine the semi-join [136] concept for distributed join processing and the filter-and-refine approach for spatial join processing. Given two sets of objects, R and S, that are at different locations (physically distributed), they send the MBRs of dataset R to the S location and filter the objects there. Then, they send the full objects from S that are in the candidate set to R's location to perform the refinement step. Tan et al. [185] point out that unlike a relational semi-join [148], where the transmission cost is dominant, the processing cost can be just as large of a factor for spatial joins.

Mamoulis et al. [128] explore the distributed spatial join problem in which the data resides on different servers between which there is no communication and the servers will not perform a spatial join. This would be the case if the servers are commercially owned and contain proprietary data. Additionally, it is assumed that the servers will not provide statistics on the data. In this case, the spatial join must be performed at a third, possibly small, site, such as a mobile device. They point out that any processing at the data server sites is relatively inexpensive compared to a mobile device. Therefore, as much work as possible should be done on the data servers, such as running queries to build useful statistics on the data. To do the spatial join, they propose a grid-based partitioning spatial join (Section 2.3.4) and use the detailed statistics to determine the best grid such that data fits within the available internal memory. The statistics are also used to identify empty regions in a dataset for which no spatial join needs to be performed, thus saving valuable transmission costs.

2.6.4 Other Techniques

There has been some interest recently in improving performance of spatial joins through the use of specialized hardware such as graphics processing units (GPUs). Such an approach has been proposed by Sun et al. [183] who suggest assigning a different color to each dataset and then letting the graphics hardware render the datasets for the screen. In this case, the frame buffer can be searched for regions containing a combination of the colors, which indicates intersecting objects. From an analytical standpoint, the algorithm will be worse because of the scan of the frame buffer, but the overall performance will be improved due to the advanced hardware. However, the drawback of this approach is that the graphics hardware typically only works on convex polygons and only a limited resolution can be achieved, thereby leading to false hits that still need to be refined. In addition, Sun et al. propose using a similar approach to determine whether two polygons intersect. In this case, the edges of one polygon are assigned one color while the other polygon's edges are colored differently.

When the dataset is not updated often and speed is critical, Günther [72] suggest the use of a spatial join index [165]. In this case, all of the intersecting objects are determined in advance and the resulting pairs of ids are stored in an index. A spatial join algorithm must be used, though, to create the original set of results pairs.

Zhu et al. [217] address the issue that at times we are not interested in all of the intersecting pairs. In particular, given two spatial datasets A and B, they are interested in obtaining the k objects from A or B that intersect the highest number of objects from the other set, termed a *top-k spatial join*. They achieve this by making use of the synchronized traversal methods from Section 2.3.1.1.

Finally, we point out that Papadias and Arkoumanis [149] take a different approach to the issue of not reporting all of the intersecting pairs. They are motivated by the desire to perform a multiway spatial join in a given amount of time due to the prohibitive expense, timewise, of performing the full multiway spatial join. Their approach searches for approximate solutions, which are defined as solutions that might violate some constraints. For instance, for a 3-way intersection, a solution with only two of the required intersections would be an approximate answer. The method they use is a combination of search heuristics, randomized algorithms, and R-tree spatial indexes [76]. In essence, a candidate pair is guessed and then improved upon using a search through the R-trees. This operation is repeated until time runs out.

2.7 Selectivity Estimation

Techniques for estimating the selectivity of a spatial join are important as an aid for analyzing spatial join algorithms, for use in query optimizers, and as a data mining tool. Günther et al. [73] have shown that along with the size of the dataset, selectivity is crucial in determining the performance of an algorithm. Query optimizers assist a database engine in determining the order of operations in a complex query. Also, selectivity estimates could play a roll in determining which spatial join algorithm to use. For instance, a very dense dataset in which nearly every pair is reported, could be computed faster using the simple nested loop join (Section 2.2.1), rather than a more complex algorithm with a large overhead. For data mining applications, selectivity estimates can give approximate answers to queries, which can be used to rule out hypotheses [27].

For uniform datasets in two dimensions, a rudimentary estimate of the selectivity for the filtering stage of a spatial join can be derived from the probability that two average sized rectangles overlap [13]. Given that both datasets are enclosed in a universe of finite area, which is represented as $TotalArea$, the probability that two rectangles with widths w_a and w_b and heights h_a and h_b intersect is:

$$\frac{(w_a + w_b) \cdot (h_a + h_b)}{TotalArea} = \frac{area_a + area_b + (w_a \cdot h_b) + (w_b \cdot h_a)}{TotalArea}, \tag{2.2}$$

where $area_a$ and $area_b$ are the areas of two rectangles [20]. To derive a selectivity estimate for a spatial join between two uniform datasets, the average sizes of the rectangles in the two datasets A and B are substituted into Equation 2.2. In this case, the selectivity is approximately:

$$\frac{\overline{area_A} + \overline{area_b} + (\overline{w_A} \cdot \overline{h_B}) + (\overline{w_B} \cdot \overline{h_A})}{TotalArea}. \tag{2.3}$$

[20]Equation 2.2 assumes that the space wraps around and does not account for the need for rectangles to be within the boundaries of the space. However, the difference is negligible if the rectangle is a small fraction of the space. Also, a minus one value is also dropped from the sums, for simplicity.

However, the situation is more complicated for non-uniform datasets. Mamoulis and Papadias [131], An et al. [9], and Belussi et al. [26] propose using a two-dimensional histogram on each dataset, which can be a grid with occupancy counts. Within each grid cell, the selectivity estimate for uniform datasets, Equation 2.3, can be applied. The results for each cell are then combined to get an overall estimate of selectivity. The finer the histogram, the more accurate the results will be. An et al. also studied sampling techniques and proposed a novel technique for estimating intersections within a grid cell by counting the number of sides and corners of rectangles within each grid cell.

Das et al. [48] present a method that provides a selectivity estimate of a spatial join for skewed datasets. The method requires one scan of the dataset, and has a provable, adjustable error bound. Each object in the dataset is represented by a fixed set of objects from a *dyadic* cover, with a total size $O(\log(n))$, where n is the size of the dataset. As each object, r, is encountered, summary statistics for each dyadic object composing r are updated. By combining these statistics in a manner that does not count particular intersections more than once, the method estimates the selectivity of the spatial join.

In another approach, working with point datasets, Belussi and Faloutsos [27] expressed the selectivity of a self-spatial join in terms of a fractal dimension. Since the data is points, the spatial join finds all pairs of points within a distance ϵ, which is expressed as $\overline{nb}(\epsilon)$, and is analogous to the average number of intersecting pairs, that is, the selectivity of the spatial join [21]. The average number of nearby, or

[21]Though, $\overline{nb}(\epsilon)$ as defined by Belussi and Faloutsos [27] is the number of points within a circle,

neighboring, points captures information about the distribution of the data. If the data is clustered, then the average number of neighbors increases. The problem, then, is to find a good estimate of the number of neighbors. To do so, Belussi and Faloutsos use the fractal *correlation dimension* D_2, which is a characteristic of the dataset, and calculate the average number of neighbors as:

$$\overline{nb}(\epsilon) = (n-1) \cdot (2 \cdot \epsilon)^{D_2}, \tag{2.4}$$

where n is the number of points. Using this equation, the selectivity of a self-spatial join is estimated as $(2 \cdot \epsilon)^{D_2}$. The correlation dimension, D_2, can be calculated in $O(n \cdot \log(n))$ time [27] or even in constant time according to Faloutsos et al. [63], who describe algorithms for efficiently calculating the correlation dimension for a given dataset and show the effectiveness of the calculation for predicting the selectivity of a self-spatial join [22].

here, their generalization to other shapes is used. In this case, a rectangle.

[22]Faloutsos et al. [63] use the formula $K \cdot r^{\mathcal{P}}$ for selectivity, instead of Equation 2.3, where r replaces $2 \cdot \epsilon$ and $\mathcal{P}$ replaces D_2, and they also use the multiplier K.

Chapter 3

Iterative Spatial Join

This chapter presents the Iterative Spatial Join algorithm for performing a spatial join on unindexed low-dimensional data. Implementing the Iterative Spatial Join requires little more than providing a structure for holding the data that intersects the sweep line, which can be found in algorithm text books [46]. Furthermore, the Iterative Spatial Join has very predictable performance characteristics, as opposed to other methods that might suffer dramatically reduced performance or fail entirely at certain amounts of available internal memory for a given dataset. Some, but not all, of these shortcomings can be reduced by tuning the parameters of the algorithms, but this requires detailed knowledge of the dataset, which might not be available or expensive to get. The Iterative Spatial Join has no such parameters and dynamically adjusts to the amount of internal memory.

Our experiments show that the Iterative Spatial Join performs as well as competing methods when there is an abundance of internal memory. When internal memory is insufficient (or conversely, when the dataset is very large), then the Iterative Spatial Join either outperforms or overcomes limitations of competing methods. In addition, our experiments show that as internal memory is reduced, the performance of the Iterative Spatial Join worsens only gradually. This is in contrast to some of the competing methods which show a dramatic drop in performance at certain memory thresholds.

In addition to introducing the Iterative Spatial Join, we also introduce the concept of *density* — that is, the fraction of rectangles in a cross-section of the plane — into the analysis of such algorithms. We show that this property of the dataset has a significant impact on the performance of spatial join algorithms, and present the results of some experiments that support this conclusion.

The rest of this chapter is organized as follows. The classic plane-sweep method is described in Section 3.1 as well as a modification to increase the number of rectangles it can process with a given amount of internal memory, attributable to Arge et al. [19]. In Section 3.2, we present and analyze the Iterative Spatial Join. Section 3.3 contains the results of experimental comparisons.

3.1 Internal Memory Plane Sweep

As we saw in Section 2.3.3, a common approach to performing spatial joins with unindexed data is to partition the data until it is of a size that can be processed using an internal memory plane sweep [160]. Our algorithm, the Iterative Spatial Join, does not partition the data, but instead, repeatedly reads the data, processing as much data as possible with each pass using a plane-sweep technique. We use a modified version of the plane-sweep algorithm [19], which can process far more data than the classical plane-sweep algorithm with a given amount of internal memory. In this section, we describe the classic plane-sweep algorithm and the modified algorithm as a background to the Iterative Spatial Join, which is presented in Section 3.2. We also provide an analysis and proof of correctness of the modified version, which will serve as the basis of the analysis and proof of correctness of the Iterative Spatial Join. In addition, we define the concept of *maximum density*, and show its importance in the analysis of the plane-sweep algorithm.

3.1.1 Description

A two-dimensional plane-sweep of a set of iso-oriented rectangles finds all of the rectangles that overlap. The algorithm has two passes. The first pass sorts the rectangles in ascending order on the basis of their left and right sides (i.e., x coordinate values) and forms a list. The second pass sweeps a vertical scan line through the sorted list from left to right, halting at each one of these points, say p. At any instant, all rectangles that intersect the scan line are considered *active* and are the only ones whose intersection needs to be checked with the rectangle associated with p. This means that each time the sweep line halts, a rectangle either becomes active (causing it to be inserted in the set of active rectangles) or ceases to be active (causing it to be removed from the set of active rectangles). Thus the key to the algorithm is its ability to keep track of the active rectangles (actually just their vertical sides) as well as to perform the actual intersection test.

To keep track of the active rectangles, the plane-sweep algorithm uses a structure (referred to as a *sweep structure*) that supports three operations needed to track the active rectangles. The first inserts a rectangle by adding it to the active set. The second (referred to as `purgeInactiveRectangles`) removes from the active set all rectangles that do not overlap a given rectangle (or line). These rectangles become inactive when the sweep line halts. The third operation searches for all active rectangles that intersect a given rectangle. Examples of structures that support these operations are a dynamic interval tree [19] [1], which is a Cartesian tree [201].

In the classical rectangle intersection problem, we are given a set of rectangles, S and we need to determine the pairs of intersecting rectangles in S. In a spatial join, we are given two sets, A and B, of rectangles, and we need to determine all pairs of intersecting rectangles in A and B — that is, for each rectangle r in A, find all of the rectangles in B intersected by r, and vice versa. To apply the plane-sweep algorithm, a sweep structure is needed for both A and B. Rectangles from A are inserted in A's sweep structure and rectangles from B are inserted in B's sweep structure. Also, a rectangle r from A will perform a search on B's sweep structure, thereby finding all the intersections with the rectangles in B, and vice versa. Note that an intersection is reported only when the second rectangle in the sorted order is inserted, thereby avoiding reporting duplicate intersections.

There are several approaches to implementing the `purgeInactiveRectangles` method. If the `purgeInactiveRectangles` operation is not performed at all, then the plane-sweep algorithm will still produce the correct set of pairs of intersecting rectangles, but it will be slow because of the constantly increasing size of the sweep

[1] The dynamic interval tree differs significantly from the interval tree [54], which also stores the endpoints of all of the intervals that are parallel to the sweep line. The search time in an interval tree is faster, but the interval tree requires that all intervals are inserted before any searching is done.

```
procedure TWO_SET_PLANE-SWEEP(RectanglesA,RectanglesB)
  rectanglesC←setMerge(RectanglesA,RectanglesB);
  rectanglesC←sortByLowerLeftCorner(rectanglesC);
  sweepStructureA←createSweepStructure();
  sweepStructureB←createSweepStructure();
  foreach r ∈ rectanglesC do
    if r ∈ RectanglesA then
      purgeInactiveRectangles(sweepStructureB,r);
      searchForIntersectingRectangles(sweepStructureB,r);
      insertRectangle(sweepStructureA,r);
    else
      purgeInactiveRectangles(sweepStructureA,r);
      searchForIntersectingRectangles(sweepStructureA,r);
      insertRectangle(sweepStructureB,r);
    endif;
  enddo;
end
```

Figure 3.1: The two-set plane-sweep algorithm for spatial joins.

structures. Therefore, `purgeInactiveRectangles` is executed in order to improve performance. One approach is to perform a `purgeInactiveRectangles` on a sweep structure before searching it. In this way, any rectangles that do not intersect the current sweep line and could not possibly intersect the current rectangle are removed from the sweep structure, thereby reducing its size and increasing the speed of the search. Other policies would be to perform the `purgeInactiveRectangles` operation on both sweep structures for each rectangle processed, whether from A or B, or only perform it on a rectangle's own sweep structure. In practice though, too many calls to `purgeInactiveRectangles` are costly and a better method is not to perform the operation for each rectangle, but, instead, wait and perform it for every n^{th} rectangle that is processed. For instance, we could apply `purgeInactive-Rectangles` on A's sweep structure only after each time that ten searches have been performed on A's sweep structure and similarly with B's sweep structure.

In addition, in order to adapt the plane-sweep algorithm to solve a spatial join, the rectangles from both sets must be processed in sorted order. The sets A and B must be merged in order to perform the sort, thereby producing a sorted list in which each rectangle is identified as being from A or B. The algorithm for this two-set plane-sweep algorithm is shown in Figure 3.1.

Below, we provide a proof of the correctness of the plane-sweep algorithm, and later use this proof as the basis for proving the correctness of the Iterative Spatial Join in Section 3.2.

Theorem 3.1.1. *Given two sets of iso-oriented rectangles, A and B, the two-set plane-sweep algorithm reports all intersections between A and B, and only reports intersections between A and B.*

Proof. Consider a rectangle r from set A as it is being processed in the *for* loop of the algorithm. At the time r is processed, the rectangles in B that occur before r in the sorted list of rectangles must have already been inserted into B's sweep structure. If any `purgeInactiveRectangles` operations were performed on B's sweep structure, then they removed rectangles appearing earlier than r in the sort order and therefore these rectangles could not possibly intersect r. Therefore, B's sweep structure must contain all of the rectangles that intersect r and which appear earlier in the sorted list of rectangles. When a search is performed on B's sweep structure with r as a parameter, these intersections will be reported. After r is inserted into A's sweep structure, all of the rectangles in B that occur after r will search A's sweep structure and report any intersections with r as long as r remains in A's sweep structure. Since r will only be removed from A's sweep structure once it cannot intersect any remaining rectangles in B, all of r's intersections with rectangles appearing in B and later in the sorted order will be reported. Therefore, all of the intersections between r and B will be reported. Since this applies to any rectangle from A and B (with A and B interchanged in the proof for rectangles from B), all of the intersections will be correctly reported. Furthermore, since by definition the search operation reports only intersections with the given rectangle, only intersections between A and B will be reported. $\square$

3.1.2 Analysis

The analysis of the plane-sweep algorithm depends not only on the size of the inputs, but also on another property that we call *maximum density*, referred to as D. The maximum density of a set of rectangles is the maximum number of rectangles intersecting a line parallel to an axis divided by the total number of rectangles in the set so that $0 \leq D \leq 1$. Assuming that there are n rectangles in the set, the maximum number of rectangles in the sweep structure is $D \cdot n$. We use the term *maximum overlap* [19] to refer to this quantity (it is also known as the *cross-section number* [75]). In addition, we make use of a related property which we call the *average density*, referred to as d, to denote the average number of rectangles in the sweep structure divided by n. For example, in some of our experiments, we used the U.S. Bureau of the Census [190] Tiger file data with $d = 0.001$, meaning that on average, 0.1% of all the rectangles intersect the sweep line at any given time.

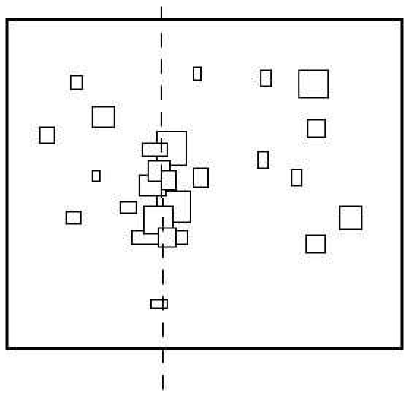


Figure 3.2: A skewed dataset containing 25 rectangles with a maximum density of 0.4. The maximum density occurs at the dashed line, where ten rectangles intersect the line.

In practice, the maximum density depends on the nature of the problem being solved and the field size (i.e., the maximum enclosing rectangle of all rectangles), as well as the number of rectangles. A common practice [75] is to use $\sqrt{n}$ as the maximum overlap (i.e., $D \cdot n$), based on observations using VLSI design data [147, 184]. While this might be true for VLSI design data, there is no reason why this would hold true in general or for data used in Geographic Information Systems (GIS). In particular, for skewed GIS data, for example, it could be that most objects lie in a small cluster, thereby creating a much higher maximum density value, D. In order to visualize how easily a dataset could violate the $\sqrt{n}$ rule, Figure 3.2 shows a simple skewed dataset with a maximum density of 0.4. There are 25 rectangles in the set, and thus, $D \cdot n = 10$, which is much greater than $\sqrt{25} = 5$.

Since D is the maximum number of rectangles encountered by a line parallel to an axis, which is a sweep line, a sweep structure will contain at most $D \cdot n$ rectangles. For the two-set plane-sweep algorithm, we need to use the D value from the set obtained by combining both sets of rectangles. The amount of internal memory required by the two-set plane-sweep will then be the amount of memory required by the rectangles in the sweep structures plus any overhead incurred by the algorithm. If *rectSize* is the number of bytes of internal memory required by a rectangle and *overhead* is the algorithm overhead in bytes, then the total amount of internal memory required by the algorithm is, in bytes:

$$\mathit{rectSize} \cdot D \cdot n + \mathit{overhead}.$$

For example, if the input is a set of rectangles with $D = 0.003$, and if each rectangle requires 40 bytes of memory, then with 10 megabytes of internal memory and an overhead of 100 KBytes, we can process 82.5 million rectangles.

3.2 Iterative Spatial Join

In this section we address the problem that occurs when there is insufficient internal memory to use the plane-sweep algorithm of Section 3.1. One solution is to revert to the methods mentioned in Section 2.3.3, which partition the original data (e.g. into a grid as in PBSM [155] or vertical strips as in SSSJ [19]) and then apply the plane-sweep algorithm to corresponding partitions, handling rectangles that overlap more than one partition separately. Instead, we propose a new method which we call the Iterative Spatial Join. The Iterative Spatial Join modifies the plane-sweep algorithm to save to external memory any rectangles it cannot immediately process. The algorithm is run repeatedly, but only those rectangles saved in external memory are inserted into the sweep structures on subsequent iterations, meaning that each rectangle is inserted into a sweep structure only once. Our method depends on an extension to the plane-sweep algorithm proposed by Arge et al. [19], in which not all of the rectangles in the datasets need to fit in internal memory. Only the sweep structures containing the active rectangles need to fit in internal memory. In the rest of this section we discuss the Iterative Spatial Join algorithm in detail. We then extend the proof of correctness and analysis given in Section 3.1 to apply to the Iterative Spatial Join. Results of an empirical comparison with partitioning methods are described in Section 3.3.

3.2.1 Description

The Iterative Spatial Join algorithm, shown in Figure 3.3, builds on the plane-sweep algorithm given in Section 3.1. When a rectangle is to be inserted into the sweep structure, we check if there is sufficient memory to insert the rectangle. If there is not enough memory, then a reference to the rectangle (e.g. an integer specifying the location of the rectangle in the sorted order) is written to a file. The algorithm continues, checking the rectangles still in the sorted list of rectangles (`insertList` in Figure 3.3) for intersections with the rectangles that have been inserted. As rectangles are removed from the sweep structure with the `purgeInactiveRectangles` operation, memory is freed and more rectangles from `insertList` can be inserted into the sweep structures again. When the sweep process (represented by the *for* loop) is finished, the file `doOverFile` contains references to all of the rectangles that were not inserted. Next, the sweep process is run again,

```
procedure ITERATIVE_SPATIAL_JOIN(RectanglesA,RectanglesB)
begin
  rectanglesC←setMerge(rectanglesA,rectanglesB);
  rectanglesC←sortByLowerLeftCorner(rectanglesC);
  sweepStructureA←createSweepStructure();
  sweepStructureB←createSweepStructure();
  insertList←rectanglesC;
  do
    initialize(sweepStructureA);
    initialize(sweepStructureB);
    doOverFile←new File();
    foreach r ∈ rectanglesC do
      if r ∈ rectanglesA then
        purgeInactiveRectangles(sweepStructureB,r);
        searchForIntersectingRectangles(sweepStructureB,r);
        if r = first(insertList) then
          insertList←rest(insertList);
          errorStatus←insertRectangle(sweepStructureA,r);
          if errorStatus = InsufficientMemoryError then
            writeRectangleToFile(doOverFile,r);
          endif;
        endif;
      else;
         // Similarly, purge and search sweepStructureA
         // and insert into sweepStructureB.
      endif;
    enddo;
    insertList←readEntireRectangleFile(doOverFile);
  while insertList ≠ ∅;
end
```

Figure 3.3: The Iterative Spatial Join algorithm.

this time inserting only rectangles from `doOverFile`, which are those that were not inserted in the previous iteration. This process is repeated until all of the rectangles have been inserted into the sweep structures. Note that the rectangles that were inserted on earlier iterations are still needed in later iterations in order to search the sweep structures, but they are not inserted into the sweep structures again.

Figure 3.4 provides an example that we use to illustrate the algorithm. Assume that internal memory can hold two rectangles (in addition to overhead). Rectangles

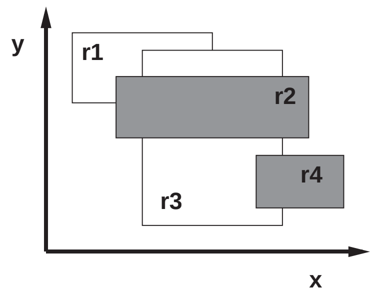


Figure 3.4: A simple example with $r1$ and $r3$ in one set and $r2$ and $r4$ in another. A correct algorithm finds the three intersections: $r1$-$r2$, $r2$-$r3$, $r3$-$r4$.

$r1$ and $r3$ make up set A, and rectangles $r2$ and $r4$ make up set B. After merging A and B and sorting, `insertList` contains all of the rectangles in order ($r1$-$r2$-$r3$-$r4$). In this example, two iterations are required. The first iteration is illustrated in Figure 3.5 and shows which rectangles are active, the position of the sweep line, the `insertList`, and the `doOverFile`. The second iteration is illustrated in Figure 3.6. The processing of each element of the *for* loop is as follows:

1. Processing starts with the first item in the dataset, $r1$. Since $r1$ is from set A, a `purgeInactiveRectangles` and `searchForIntersectingRectangles` are performed on `sweepStructureB` with no effect as `sweepStructureB` is empty. Next, $r1$ is inserted into `sweepStructureA`, thus becoming active. Figure 3.5a shows the state of the algorithm at this point.

2. Using $r2$, a `purgeInactiveRectangles` is done on `sweepStructureA` with no effect since $r1$ is still active, but the search of `sweepStructureA` reports the intersection between $r1$ and $r2$. $r2$ is inserted into `sweepStructureB`. Now,

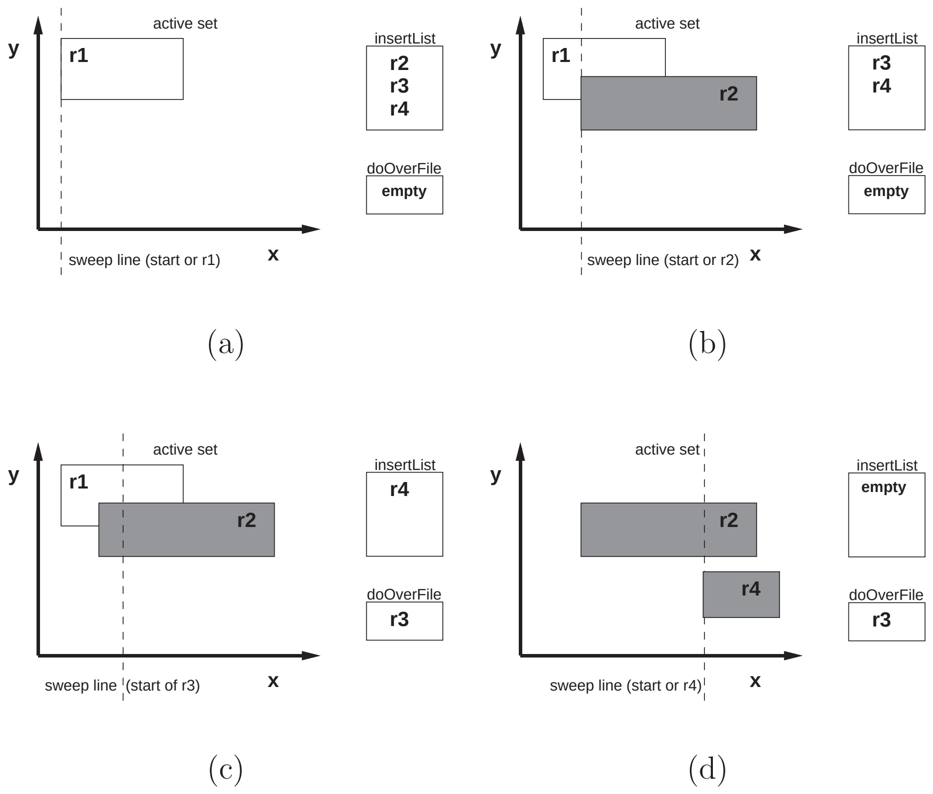


Figure 3.5: The active set, sweep line, insert list and `doOverFile` in the first iteration of the algorithm for the data of Figure 3.4 immediately after encountering the left boundary of rectangles (a) $r1$, (b) $r2$, (c) $r3$, and (d) $r4$ while processing `RectanglesC`.

the internal memory (active set) is full with $r1$ and $r2$. Figure 3.5b shows the state of the algorithm at this point.

3. Using $r3$, the algorithm performs the `purgeInactiveRectangles` method on `sweepStructureB`, producing no deletions. Next, the algorithms reports the intersection between $r2$ and $r3$ when the `searchForIntersectingRectangles` method is performed on `sweepStructureB` using $r3$. When the algorithm tries to insert $r3$ into `sweepStructureA`, an insufficient memory error is generated

and a reference to $r3$ (the integer 3) is written to `doOverFile`. Figure 3.5c shows the state of the algorithm at this point.

4. Using $r4$, the `purgeInactiveRectangles` call on `sweepStructureA` causes $r1$ to be removed because it is no longer active. Since `sweepStructureA` is empty, the search on `sweepStructureA` reports no intersection. Now there is enough memory to insert again and $r4$ is inserted into `sweepStructureB`. Figure 3.5d shows the state of the algorithm at this point.

The first *do* loop ends with $r3$ yet to be inserted. At this point, the sweep structures are reinitialized and the contents of `doOverfile` are read into `insertList`, which consists of only $r3$. In the second iteration, all of the items are again read from the dataset, but they are processed differently.

1. Using $r1$, the `purgeInactiveRectangles` method produces no deletions and the `searchForIntersectingRectangles` method reports no intersections. Notice that $r1$ is not inserted into a sweep structure since it is not in `insertList`. Figure 3.6a shows the state of the algorithm at this point.

2. Similarly, $r2$ produces the same result as for $r1$ with `sweepStructureA` and thus $r2$ is not inserted. Figure 3.6b shows the state of the algorithm at this point.

3. Both the `searchForIntersectingRectangles` and the `purgeInactiveRectangles` methods are performed on `sweepStructureB` using $r3$ with no effect. Then, $r3$ is inserted into `sweepStructureA`. Figure 3.6c shows the state of the

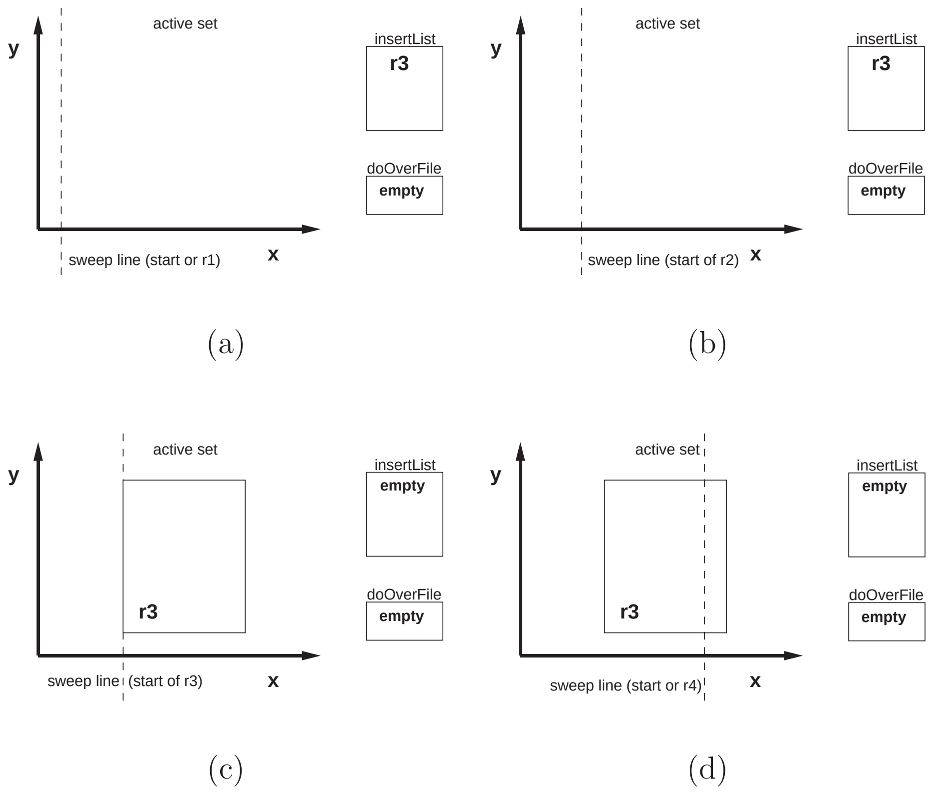


Figure 3.6: The active set, sweep line, insert list and `doOverFile` in the second iteration of the algorithm for the data of Figure 3.4 immediately after encountering the left boundary of rectangles (a) $r1$, (b) $r2$, (c) $r3$, and (d) $r4$ while processing `RectanglesC`.

algorithm at this point.

4. When $r4$ is encountered, `sweepStructureA` is searched and the intersection between $r3$ and $r4$ is reported. No rectangles are purged with the method `purgeInactiveRectangles`, and $r4$ is not inserted. Figure 3.6d shows the state of the algorithm at this point.

Since now `insertList` is empty, the *for* loop and *do* loop end, having reported all three intersections in two passes.

Now let us prove the correctness of the Iterative Spatial Join algorithm.

Theorem 3.2.1. *Given two sets of iso-oriented rectangles, A and B, the Iterative Spatial Join algorithm reports all intersections between A and B, and only reports intersections between A and B.*

Proof. The proof of correctness is similar to the proof for the two-set plane-sweep algorithm given for Theorem 3.1.1. To extend that proof, we must account for the multiple iterations. Again consider a rectangle r from set A as it is being processed in the *for* loop of the algorithm. In some iteration of the *for* loop, at the time r is processed, all of the rectangles in B that occur before r in the sorted list of rectangles must have been inserted into B's sweep structure. If any `purgeInactive-Rectangles` operations were performed on B's sweep structure, then they removed rectangles appearing earlier than r in the sort order and therefore these rectangles could not possibly intersect r. Therefore, B's sweep structure contains all of the rectangles that intersect r and which appear earlier in the sorted list of rectangles in some iteration. When the searches are performed on B's sweep structure in each iteration with r as a parameter, these intersections are reported. After r is inserted into A's sweep structure, all of the rectangles in B that occur after r will search A's sweep structure in the same iteration and report any intersections with r as long as r remains in A's sweep structure. Since r will only be removed from A's sweep structure once it cannot intersect any remaining rectangles, all of r's intersections with rectangles appearing in B and later in the sorted order are reported. Therefore, all of the intersections between r and B will be reported. Since this applies to any rectangle from A and B (with A and B interchanged in the proof with rectangles from B), regardless of the iteration, all of the intersections will be correctly reported. Furthermore, since by definition the `searchForIntersectingRectangles` operation reports only intersections with the given rectangle, only intersections between rectangles in A and B will be reported. $\square$

3.2.2 Analysis

The Iterative Spatial Join performs the same number of insertions and deletions into the sweep structures as would the two-set plane-sweep algorithm if it had sufficient internal memory. Each rectangle is inserted into the sweep structure once and removed once. The only difference in performance is the number of searches. For each rectangle, one search is performed on each iteration. This decreases the performance compared to an internal memory plane-sweep, which performs just one iteration, and thus, the sweep structure is searched just once for each rectangle. However, this decrease is mitigated by the smaller size of the sweep structure, since it is limited by the amount of internal memory.

As will be shown, the Iterative Spatial Join algorithm requires $\lceil \frac{D \cdot n}{p} \rceil$ iterations, where p is the number of rectangles that fit in the sweep structures (excluding overhead) and D is the maximum density (defined in Section 3.1.2). If there is sufficient internal memory for the internal plane-sweep algorithm to run, then the sweep structure will hold $D \cdot n$ rectangles at some point. If $D \cdot n$ is greater than p, then we need to run procedure Iterative Spatial Join again, and the sweep structure will contain p rectangles at this point on the first pass. The algorithm will try to fit the remaining rectangles in the sweep structure on the next pass, or later passes if necessary. In each additional pass, the sweep structure will contain a maximum of p rectangles until the final pass. This results in the number of iterations being $\lceil \frac{Dn}{p} \rceil$.

The algorithm reads the entire input stream for each iteration. It also writes an integer (or some other reference to a rectangle) for every rectangle not entered in the sweep structure on each iteration, and reads it in the subsequent iteration. In the worst case, each iteration could result in nearly the same number of integer reads and writes as the number of rectangles (e.g., if internal memory can only hold one rectangle), but, in practice, this number is just a fraction of the input size.

3.3 Experimental Results

We tested the algorithms using spatial datasets (Sequoia [182] and U.S. Bureau of the Census [190] Tiger data). These datasets have a low maximum density and modest amounts of internal memory are sufficient to process the data using the internal memory plane-sweep algorithm. The low density datasets make testing the external-memory algorithm more difficult. In order to adequately test the external-memory algorithm, we generated higher density datasets and limited the amount of internal memory available to the algorithm. As noted in Section 3.1, the density of a dataset also has an impact on the amount of internal memory required, and we varied this quantity as well. In the experiments, we compared the Iterative

Spatial Join, PBSM [155], SSSJ [19], and S3J [116] methods. The rest of this section is organized as follows. Section 3.3.1 describes the experimental setup and implementation issues, while Section 3.3.2 describes the data we used. Section 3.3.3 presents the results of the experiments.

3.3.1 Experimental Setup and Implementation

We implemented all four algorithms in the Sand environment [58], coding in C++. The experiments were performed on a Sun Ultra-5 SparcStation running SunOS 5.7. Within the Sand environment, we track the amount of internal memory used and can limit it. Thus, the amount of internal memory is configurable and is used as a parameter in the experiments.

We used the algorithms for PBSM, SSSJ, and S3J described in Section 2.3.3 with only minor modifications. We improved the performance of SSSJ by decreasing the overhead of the partitions (the size in bytes of the C++ partition class); we used replication for S3J; and we removed the need for duplicate elimination from PBSM.

We originally used an implementation of the SSSJ in TPIE (Transparent Parallel I/O Programming Environment) [199]. In moving to the Sand environment, we reduced the memory requirements for the strips (partitions) and auxiliary partitions (used during the partitioning phase) by making the code less general, and thus, dramatically improved the performance because many more strips could be produced during the SSSJ partitioning phase.

We used object replication in S3J as recommended by Dittrich and Seeger

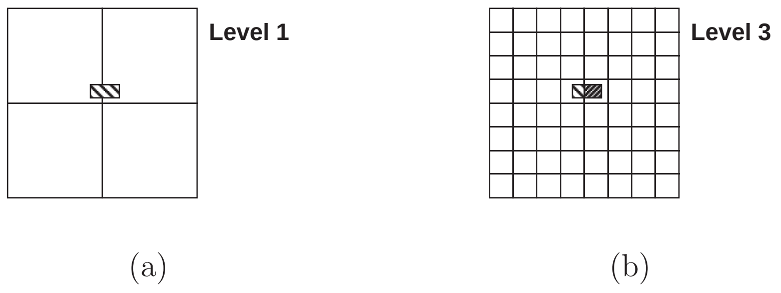


Figure 3.7: (a) Without replication, the small rectangle will be put into the zero level file meant for larger objects since it intersects the partition of level file 1. (b) Using object replication, the same rectangle is divided and put into the level file appropriate for its size, level 3.

[50], who show that small objects that intersect partition boundaries will be put into partitions meant for larger objects, as shown in Figure 3.7a. Since partitions in the upper level files will be repeatedly joined with corresponding partitions in lower level files, adding extra objects will impair performance because of the larger sizes of the upper level files. By replicating objects, as in Figure 3.7b, the upper level files become smaller, which improves performance more than object duplication hinders performance.

Also, as mentioned by Dittrich and Seeger [50], because the S3J algorithm creates such a large number of partitions, many partitions will contain only a few objects, especially for the lower level files. In this case, a nested loop join (i.e. comparing each rectangle in set A to each rectangle in set B) is the most efficient processing method for joining two of the partitions. For larger partitions, we used the plane-sweep method. We experimentally determined what constitutes a large partition in this case and determined that if the combined sizes of the two partitions is less than 6,000 objects, then it is best to use a nested loop join.

Our implementation of the filter tree that is used by S3J partitions the data into external memory files, one for each level of the tree, each of which is sorted using a space-filling curve order such as a Morton order. The sort is in terms of creating an additional MBR b_f for each MBR b of the original data where b_f is the minimum enclosing quadtree block. Each b_f is represented by its Morton code which is then subjected to a two-level sort where the primary key is the level of the block (i.e., depth in the tree), and the secondary key is the actual Morton code of the block (i.e., the bit interleaved coordinate values of one of its corners). This results in a partition of the MBRs by level which increases the likelihood that the partition is small enough to fit in memory so that we can perform a faster in-memory sort rather than an external memory sort. During the join, a block from each partition is kept in memory. In this way, each partition file is read sequentially once. The need to construct the Morton code for each block as well as sorting the blocks constitutes the overhead of the S3J algorithm.

We eliminated duplicate detection from PBSM using a known method [12, 14, 50]. In particular, the overlap of two intersecting rectangles forms a rectangular region. An intersection is only reported if the lower left corner of the overlapping rectangular region is within the current partition. Special attention needs to be given to cases where this point lies on the border of a partition. In this case, partitions are considered open on two adjacent sides.

The PBSM algorithm requires tuning of its tiling parameters. When the algorithm initially partitions, it creates a tiling of the data space and then merges these tiles into partitions. Any object that overlaps a tile is inserted into the tile's partition, leading to replication of objects. Ideally, the algorithm creates partitions that fit into internal memory. However, because the algorithm duplicates objects, the size of a partition cannot be predicted. If a partition has more objects than fit in internal memory, the PBSM algorithm must divide the partitions that are too large. This repartitioning degrades performance. The tiling can be tuned to accommodate the data, though. We found that using 16x16 tiling achieved the best performance in our experiments by trial and error. We are not aware of any mechanism for automatically tuning the tiling, but such a mechanism might involve a costly analysis of the dataset. Even if such a mechanism were to exist, in the worst case scenario, if only n objects fit in internal memory and more than n objects overlap a point, then no partitioning exists because PBSM replicates the data. In this instance, the algorithm will fail, regardless of tuning.

We ignored one case where tuning is necessary. In particular, in the original description of PBSM [155], the tiles, which form a grid, are assigned to the partitions by using a hash function or in a round robin fashion, as shown in Figure 3.8a. Experiments [155] showed that the round-robin method performed as well as or better than the hash function method. However, when using the round-robin method, if the number of partitions (determined by the amount of available internal memory) is a multiple of the number of tiles in a grid row, then the partition becomes a strip, as shown in Figure 3.8b. However, unlike SSSJ, the strip is parallel to the sweep line and the sweep algorithm will perform poorly, or, if there is insufficient memory, repartitioning will be required. Regardless of this shortcoming, we used

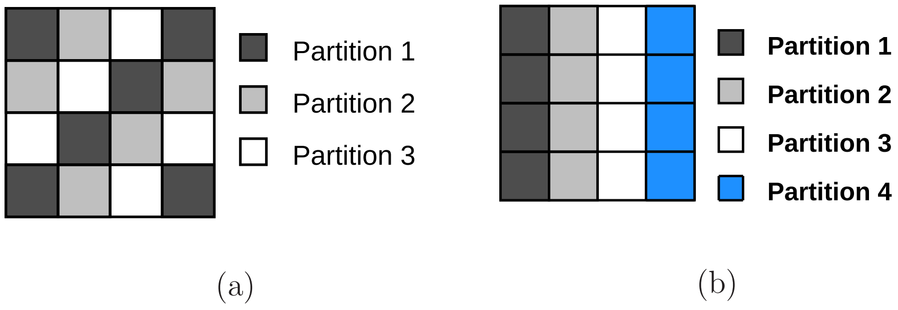


Figure 3.8: A tiling, as used by PBSM, of space into a 4x4 grid. The tiles are assigned to the partitions in a round-robin order. In (a) three partitions are created and in (b), four partitions are created.

the round-robin method since it was shown to be generally superior. We did not devise a scheme to overcome this shortcoming. However, we do point out where this occurs in the results.

3.3.2 The Data

In the first experiments, we used the U.S. Bureau of the Census [190] Tiger File data and Sequoia Benchmark data [182]. From the Tiger File data we used road and water dataset with 200,000 and 37,000 objects respectively. We used the Sequoia point data and polygon data (excluding interior holes in the polygons) with 62,556 and 58,586 objects, respectively. Both these datasets have low maximum density and can be processed using a plane sweep without having the Iterative Spatial Join and SSSJ algorithms resort to external memory.

To test the external memory algorithms, we generated random datasets and reduced the amount of available internal memory in order to provide sufficient conditions for the external-memory algorithm to be invoked. We used a random rectangle generator to create datasets with a desired average density and maximum density. Figure 3.9 shows an example dataset that we generated with 100 rectangles and a density of 0.12. We can vary the density because it is related to the size of the rectangles as a proportion of the space from which the rectangles are drawn. To see this, let r be the width of a rectangle in the sweep direction expressed as a fraction of the space. The probability that this rectangle intersects a sweep line is r. If we have n independently generated rectangles and assuming that the space wraps around (i.e. it is continuous), then the number of rectangles intersecting a sweep line is $n \cdot r$. Since we have also defined the number of rectangles intersecting a sweep line to be $d \cdot n$, then $d = r$ means that r has the same value as the density. Note that the space from which the rectangles are drawn does not normally wrap around in a 2-dimensional map, thereby causing the density at the beginning and the end of the plane-sweep to lessen, which slightly changes the actual average density, but not significantly.

3.3.3 The Results

3.3.3.1 Real Data

We compared the Iterative Spatial Join, SSSJ, PBSM, and S3J algorithms using U.S. Bureau of the Census [190] Tiger File data and the Sequoia benchmark data [182]. The results are shown in Figures 3.10 and 3.11, respectively. The PBSM algorithm performed best and the most consistently. It was not effected by the

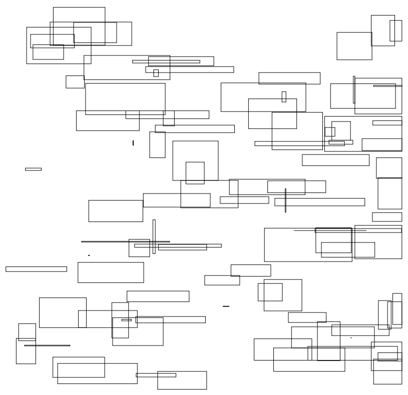


Figure 3.9: An example of the randomly generated dataset. This set has 100 rectangles and a density of 0.12.

amount of internal memory. The other three algorithms performed worse due to the need to sort. Figures 3.10a and 3.11a show the total time and Figures 3.10b and 3.11b show the actual join time without the initial sort or filter tree creation, revealing that the actual join time is generally constant for all of the algorithms. Figures 3.10b and 3.11b do not include the PBSM because it immediately partitions the data and begins processing, thereby having little initial overhead.

The differences between the performance of the algorithms can be explained by the differences in sort times and the creation of the filter trees. The Iterative Spatial Join and SSSJ algorithms do not need to use their external memory algorithms and only perform a plane sweep after sorting the data. Thus, the only difference between these two algorithms and the PBSM algorithm is that the PBSM algorithm partitions the data first, then sorts each partition separately before performing a plane sweep. Because the partition for the Iterative Spatial Join and SSSJ algorithms is bigger than the PBSM partitions, the sort routine we use must use external memory,

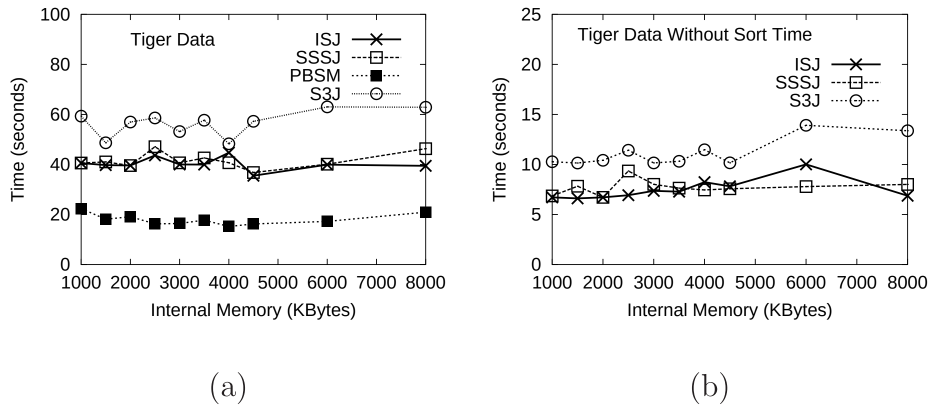


Figure 3.10: The performance of the Iterative Spatial Join (ISJ), SSSJ, PBSM, and S3J algorithms on Tiger File data (in seconds) for a) total time and b) total time without sort time (excludes PBSM data).

and therefore, impairs performance. The point at which the sort routine requires the use of external memory can be seen in Figure 3.11a to be when the memory is less than 6000 KBytes, and the Iterative Spatial Join and SSSJ algorithms no longer perform as well as the PBSM algorithm. A similar problem occurs for the S3J algorithm. With less memory, the creation of the filter trees for the S3J algorithm took more time, even though the actual join time was less than the sort time. As shown in Figures 3.10b and 3.11b, though, the S3J algorithm performs slightly worse than the Iterative Spatial Join and the SSSJ algorithms even when sort times and overhead are ignored.

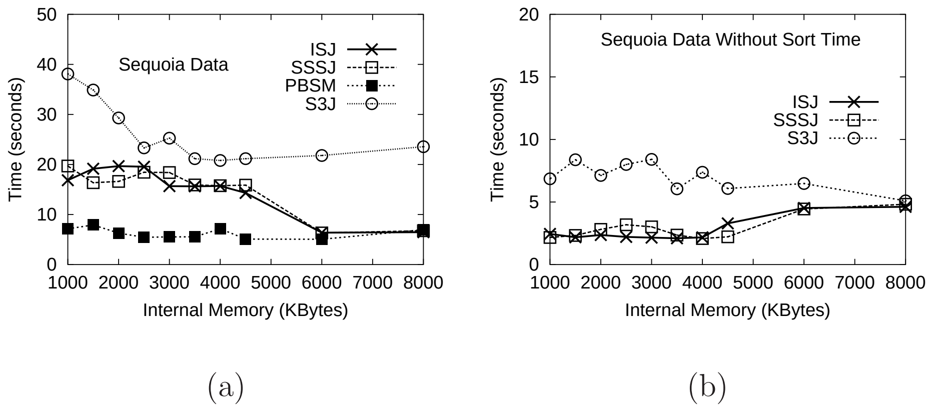


Figure 3.11: The performance of the Iterative Spatial Join (ISJ), SSSJ, PBSM, and S3J algorithms on Sequoia data (in seconds) for a) total time and b) total time without sort time (excludes PBSM data).

3.3.3.2 Experiments Varying Available Internal Memory

As mentioned in Section 3.3.2, with smaller datasets, it is difficult to experiment with external-memory algorithms. Most computers today have enough memory to process smaller datasets in internal memory. We wished to investigate the performance of the algorithms on larger datasets. One approach would have been to use all of the available internal memory and use very large datasets. As shown in Section 3.1, the internal plane-sweep algorithm is capable of handling millions of rectangles on a typical computer. Such large datasets are beyond the limits of our computers and would take excessive amounts of time for each experiment. Instead, we limit the amount of internal memory available to the algorithm and use smaller datasets.

Our experiments show that when external memory is necessary, the Iterative

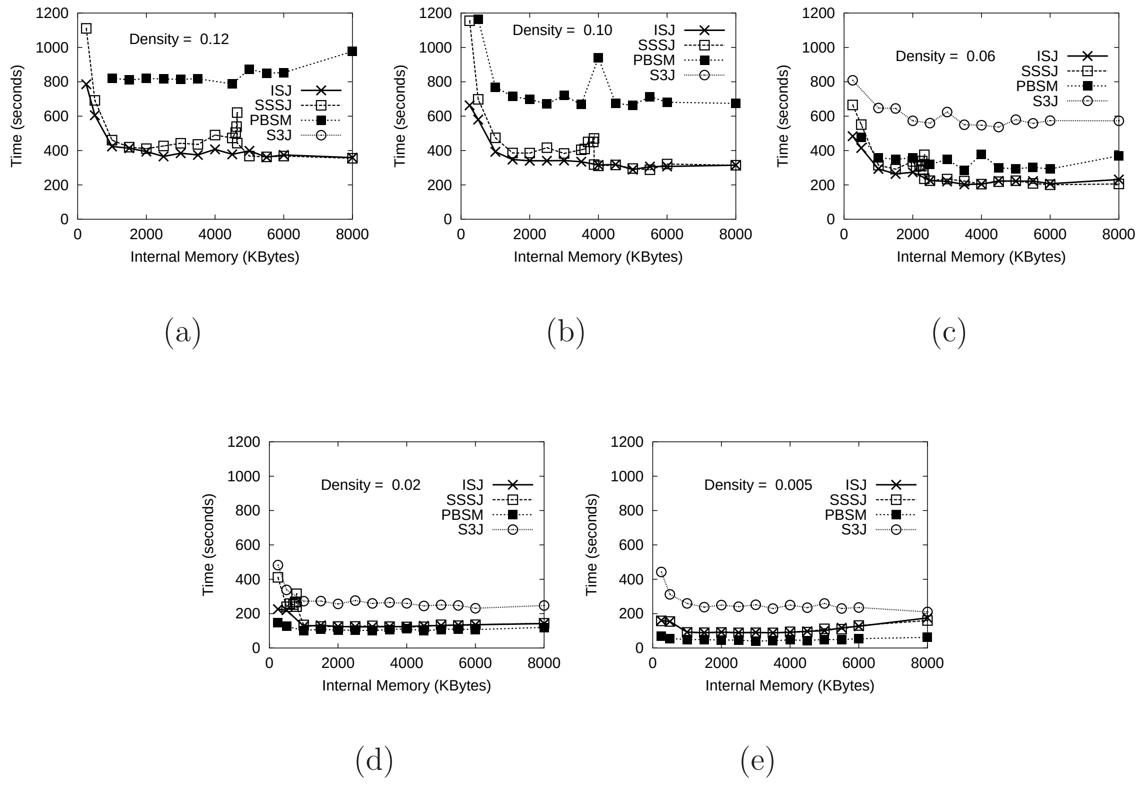


Figure 3.12: The performance (in seconds) of the Iterative Spatial Join (ISJ), SSSJ, PBSM, and S3J algorithms for varying limits of internal memory. The number of rectangles was held fixed for all experiments at 300,000. Each graph represents a set of experiments performed with different densities : (a) 0.12 (b) 0.10 (c) 0.06 (d) 0.02 (e) 0.005.

Spatial Join and SSSJ algorithms perform better than the PBSM and S3J algorithms, as shown in Figure 3.12. For higher densities, the performance of the S3J algorithm degrades so significantly that it is not shown in Figures 3.12a and Figures 3.12b. These two figures are duplicated with a larger scale in Figures 3.13a and Figures 3.13b to include the S3J algorithm.

For all densities, the Iterative Spatial Join and SSSJ algorithms perform similarly, except at the point where there is insufficient internal memory and external memory is required. At this point, the SSSJ algorithm has a spiked increase in execution time. This occurs at around 4500 KBytes in Figure 3.12a, 3500 KBytes in Figure 3.12b, 2500 KBytes in Figure 3.12c, and 800 KBytes in Figure 3.12d. This spike does not occur in Figure 3.12e since the algorithms do not need to use external memory. This spike occurs because, as SSSJ resorted to its external memory algorithm, it discarded previous results, repartitioned, and started over. This is also the point where the Iterative Spatial Join uses its external memory algorithm. For less memory, the SSSJ algorithm fails earlier, and loses less ground. This is illustrated in Figure 3.12 where the performance improves after the spike.

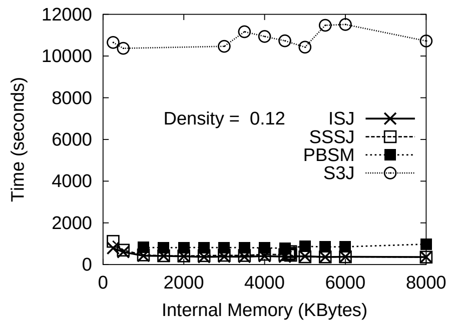
As mentioned in Section 3.3.1, we did not tune PBSM for the case when the number of partitions is a multiple of the number of tiles. The effect is decreased performance, which can be seen in Figures 3.12b and 3.12c at 4000 KBytes. This data point is not represented in Figure 3.12a because the basic PBSM algorithm failed and repartitioning was required. We did not implement repartitioning as this was beyond the scope of the work, although a repartitioning mechanism does exist [50]. The 500 KByte point for PBSM is also missing from Figure 3.12a because repartitioning is required.

We also wished to explore the effects of varying the average density on the performance of the algorithms. As shown in Figure 3.12, the density had two effects. The first effect is longer running times. Higher density data means that the plane-sweep algorithm will have more objects in its sweep structures, reducing its performance and the overall running time. The second effect is to change the point where the Iterative Spatial Join and SSSJ algorithms require the use of their external memory algorithms. As predicted in Section 3.1, for higher densities, the internal-memory plane-sweep algorithm used by both SSSJ and Iterative Spatial Join requires more internal memory, and the algorithms will need to use their external memory algorithms for larger amounts of given internal memory. For example, in Figure 3.12b, the spike indicates that the Iterative Spatial Join and SSSJ algorithms are using external memory when internal memory is less than 3500 KBytes. However, when the density is greater, as in Figure 3.12a, the algorithms will need to use external memory if internal memory is less than 4500 KBytes.

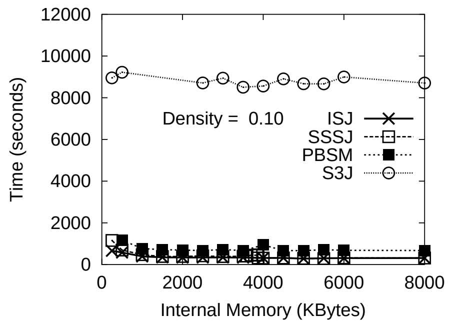
For the higher densities, the S3J algorithm performed poorly. The higher density data contains a greater proportion of larger objects, which will fill the upper S3J level files. The partitions in these level files are compared to the many corresponding partitions in the lower partition files. Joining these partition files will be quick, but the shear number of these pairings significantly degrades performance. The data points in Figure 3.13 for S3J are missing around 2000 KBytes because the partitions created were too large and needed to be repartitioned. The S3J algorithm, however, has no mechanism for repartitioning its data. Interestingly, with less internal memory (250 and 500 KBytes), the S3J algorithm partitioned the data differently and thus was able to process the data completely.

3.3.3.3 Experiments Varying the Input Size

Figure 3.14 illustrates a typical set of results for a higher density dataset with a large number of rectangles. In both experiments, we fixed the average and max-



(a)

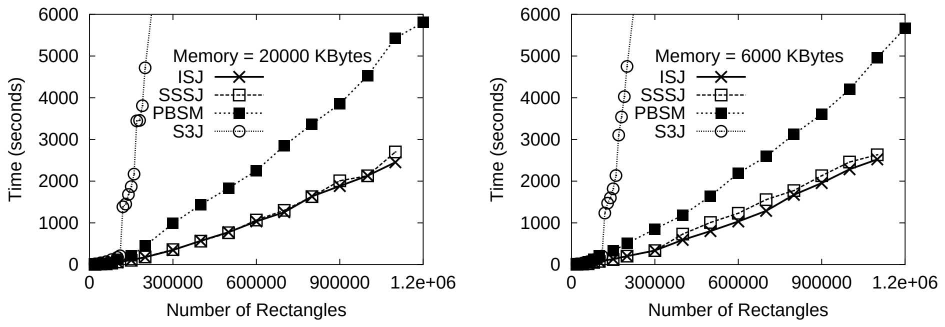


(b)

Figure 3.13: Expansion of the scale of Figure 3.12 to include all algorithms. The performance (in seconds) of the Iterative Spatial Join (ISJ), SSSJ, PBSM, and S3J algorithms for varying limits of internal memory. The number of rectangles was held fixed for all experiments at 300,000. Each graph represents a set of experiments performed with different densities: (a) 0.12 (b) 0.10.

imum density at 0.12 and fixed the available amount of internal memory. We then varied the number of rectangles in the input dataset. In both cases, the Iterative Spatial Join and SSSJ algorithms behaved similarly and exhibited better performance than the PBSM algorithm. The S3J algorithm performed poorly for the reasons mentioned in Section 3.3.3.2

The decreased performance spike of the SSSJ algorithm described in Section 3.3.3.2 is not shown on these graphs because of the fewer number of data points represented on these graphs. The spikes occur between the data points shown.



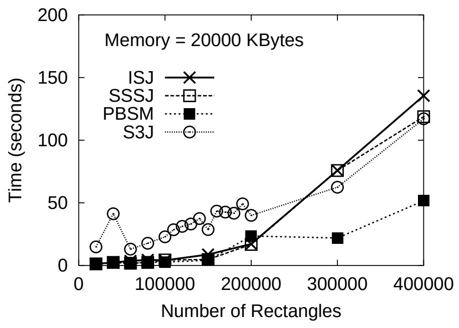
(a) (b)

Figure 3.14: The performance (in seconds) of the Iterative Spatial Join (ISJ), SSSJ, PBSM, and S3J algorithms for varying input sizes (number of rectangles in each dataset). The maximum and average density was held fixed for all experiments at 0.12. Each graph represents a set of experiments performed with different limits to internal memory: (a) 20,000 KBytes (b) 6000 KBytes.

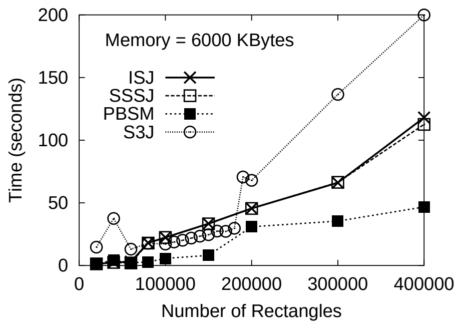
3.3.3.4 Algorithm Overhead

Before performing the spatial join, each of the algorithms partitions the data, sorts the data, or even does both. Figure 3.15 separates this overhead cost from the total execution time given in Figure 3.14. In particular, the overhead cost encompasses all the preprocessing that takes place before starting the execution of the actual spatial join algorithm. For example, the PBSM algorithm partitions the data first. For small datasets, it is possible that only a single partition is necessary as can be seen in Figure 3.15, in which case the overhead cost is negligible. The Iterative Spatial Join and SSSJ algorithms sort the data first, while the S3J algorithm creates a filter tree by partitioning the data into level files and then sorting each level file using a space-filling curve (we use a Morton order). The Iterative Spatial Join and

SSSJ algorithms incur the same overhead cost since they both execute the same sorting algorithm. The S3J algorithm performs worse in some cases and better in other cases in comparison to the Iterative Spatial Join and SSSJ algorithms because the sizes of the level files change for different input sizes and have a large effect on the performance (e.g., skewed files will effect performance). Comparing Figure 3.15 with Figure 3.14, we see that the overhead cost is a small percentage of the overall cost for this data set with a higher density. For lower density data, the overhead cost can dominate the cost of the join. As can be seen in Figure 3.10, the total cost of the spatial join (Figure 3.10b) is the lesser part of the total join cost (Figure 3.10a). Also, note that the overhead cost is effected by the amount of available internal memory. In particular, as the amount of available internal memory increases, as in Figure 3.15a, the overhead cost decreases as in Figure 3.15b.



(a)



(b)

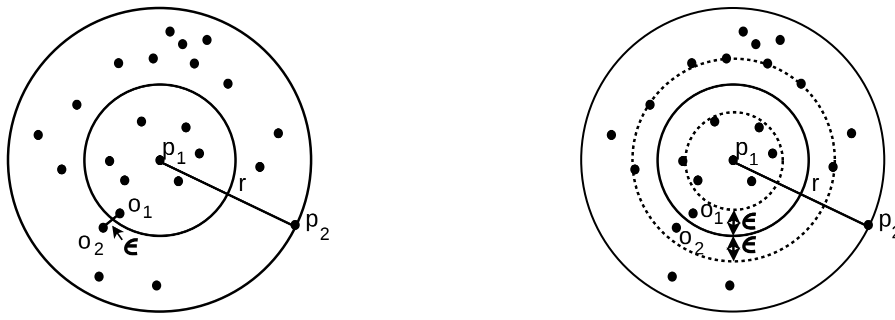
Figure 3.15: The overhead cost (in seconds) of the Iterative Spatial Join (ISJ), SSSJ, PBSM, and S3J algorithms for varying input sizes (number of rectangles in each dataset). The maximum and average density was held fixed for all experiments at 0.12. Each graph represents a set of experiments performed with different limits to internal memory: (a) 20,000 KBytes (b) 6000 KBytes.

Chapter 4

Quickjoin

This chapter describes the Quickjoin algorithm for performing a similarity join. It is based on the Quicksort algorithm [86] in that it recursively partitions the data until each partition contains a few objects, at which point a nested-loop join is used. The data can be partitioned using a variety of techniques, such as *ball partitioning*, which partitions the data based on their distance to a single random object (termed a *pivot*) [186, 211], as shown in Figure 4.1, or *generalized hyperplane partitioning*, which partitions the data on the basis of which of two randomly selected objects is closest [188]. Both techniques, as well as other variations, are described in Section 4.2.

By just partitioning the data, a significant portion of the desired results will be missed since objects in one partition might need to be joined with objects in another partition. For example, using ball-partitioning, in Figure 4.1a, objects o_1 and o_2, which lie within ϵ of each other, would not be reported as similar since they are in different partitions. To overcome this problem, when the data is partitioned into two partitions, the data is replicated into two sets: set S_l with distance values from p_1 less than or equal to the partitioning distance, r, but within ϵ of r, and set S_g with distance values from p_1 greater than r, but also within ϵ of r, as shown in Figure 4.1b. These two "windows" are joined, and only pairs, (a, b), with an object



(a) (b)

Figure 4.1: (a) With ball partitioning, objects are partitioned by their distance from a randomly chosen object p_1, based on a distance r. In this example, r is determined using half the distance to the farthest object p_2. The problem with this partitioning is that it can separate result pairs that are within ϵ of each other, such as objects o_1 and o_2. (b) In order to correctly capture all result pairs, two windows of objects within ϵ of the partitioning distance, r, are created – one with objects whose distances from p_1 are greater than r and one with objects whose distances from p_1 are less than or equal to r.

from each window are reported, where $a \in S_l$ and $b \in S_g$ or $b \in S_l$ and $a \in S_g$. Again, a Quicksort-like partitioning is applied recursively, but this time to each of the windows.

The rest of the chapter is organized as follows. After reviewing related work in Section 4.1, the details of the algorithm are presented in Section 4.2, including additional variations of the algorithm. In Section 4.3, the performance characteristics of the algorithm are analyzed in terms of CPU, internal memory, and I/O costs. Section 4.4 describes the result of several experiments showing that the method is competitive compared to the EGO method [31], described in Section 4.1.1 and the GESS method [51], described in Section 4.1.2.

4.1 Related Work

A similarity join is similar to a spatial join [145] or an interval join [57], which find pairs of intersecting objects, such as intervals, polygons, or rectangles, in low dimensions. Algorithms have been devised for spatial joins using both indexed and unindexed methods, as described in Chapter 2. For indexed methods, the join algorithm is frequently a synchronized traversal of the tree-like index structures [38]. If the data is not indexed, then an index can be built, or if the index will not be reused, then the expense of building an index can be avoided by using a partitioning method [155] or a plane-sweep method [19, 98]. As with the spatial join techniques, the first attempts at solving the similarity join used indices [117, 178]. Unfortunately, traditional spatial indices, such as the R-tree [76], have a number of drawbacks in higher dimensions [32, 177, 178]. For example, the number of neighboring leaf nodes typically increases in higher dimensions, thereby degrading performance. To overcome such drawbacks, several indices have been proposed for indexing high-dimensional data in order to perform a similarity join:

1. An MX-CIF quadtree [107] that uses a multi-dimensional linear order [51, 117]. One variant, the GESS method [51] is described in more detail in Section 4.1.2.

2. A variant of the kdb-tree [163] that splits the data in each dimension into ϵ sized segments [177, 178].

3. A sophisticated R-tree variant with large pages and a secondary search structure [32].

4. A simple grid-like index structure [103], similar to a grid file [140].

Once the data is indexed, the similarity join can be performed using either a synchronized tree traversal [38] or other heuristic methods that globally order the access of the data pages [78, 102, 110].

However, as mentioned in the chapter introduction, similarity join techniques based on these indices might not perform well since they rely on indexing a subset of the dimensions and it might be difficult or impossible to choose a good subset of dimensions. Furthermore, these techniques assume a fixed, Minkowski distance function, which limits their applicability. For example, Shim et al. [177, 178] propose an index structure, termed an ϵ-kdB tree, in which each level of the tree splits on a different dimension into approximately ϵ sized pieces. After the indices are built, a synchronized tree traversal is used to perform the join. This method relies on choosing a good split dimension, which might not exist for larger ϵ values. Similarly, two good dimensions must be chosen for the *Grid-join* method of Kalashnikov and Prabhakar [103], which uses a 2-dimensional grid to index multi-dimensional data. Even in their work, they noted that this approach is unsuitable for higher dimensions. In their experimental section they point out that "the Grid-join is not competitive for high-dimensional data, and its results are often omitted for clear presentation of the EGO-join and EGO*-join results."

Instead of creating a new index, Böhm et al. [31] propose an unindexed method, termed the *Epsilon Grid Order* or EGO, which is described in detail in Section 4.1.1. While the EGO method was shown to outperform traditional indices, some evi-

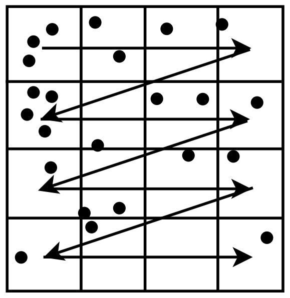


Figure 4.2: The EGO sorted order traverses grid cells. Points within a grid cell are unsorted.

dence [102] suggests that the EGO method does not outperform the indices that are tailored to higher dimensions when the indices are already built. In our own experiments, given in Section 4.4, we have found that the Quickjoin method and the EGO method both significantly outperform one of the indexed methods, the GESS method [51].

4.1.1 Epsilon Grid Order (EGO)

The EGO algorithm of Böhm et al. [31] is based on the EGO sorted order, which is described as a sort based on imposing an ϵ-sized multi-dimensional grid over the space, and ordering the cells, as shown in Figure 4.2. In effect, this is a loose version of a total multi-dimensional sort, where the points are primarily sorted in one dimension, secondarily on another dimension, and so forth for each dimension.

In the EGO method, the EGO-sorted data is read in blocks, and the key to the algorithm is to efficiently schedule reads of blocks of data so as to minimize I/O. Each block b of data needs to be joined with all other data blocks containing data within a distance of ϵ of b. In the best case, each data block can be read one at a time, and all corresponding data blocks (those that need to be joined) will fit in memory. In this way, each data block will be read only once. For larger ϵ values, this will not likely be the case, and the problem becomes similar to the classic join scheduling problem, which is known to be NP-Hard [135, 139]. To overcome this problem, Böhm et al. use heuristics to order the block reads and to determine which blocks in memory to evict. The goal of the heuristic, which they term *crab stepping*, is to try to minimize repeated reads of blocks.

To join two blocks of data, the data blocks are recursively divided until a minimum size has been reached, in which case a nested-loop join can be used. The recursion also stops when the data block subsets are at a distance greater than ϵ from each other. With the EGO* algorithm, Kalashnikov and Prabhakar [103] use a better test for checking for this condition, which stops the recursion sooner and improves performance.

4.1.2 GESS

Dittrich and Seeger created the *Generic External Space Sweep* (GESS) method [51], which is a variant of the *Multidimensional Spatial Join* (MSJ) of Koudas and Sevcik [117, 118], which in turn is based on a technique for performing a spatial join on rectangles [116]. The technique is adapted to multi-dimensional points by creating hyper-squares centered around each point with ϵ length sides, and then using the original technique to join the hyper-squares. If two hyper-squares intersect, then the points must be within ϵ of each other along at least one of the dimensions of the underlying space. The GESS and MSJ methods assign the data to grid cells, sort the data based on the assigned grid cells using a multi-dimensional linear order [99, 170], and then scan the data in sorted order to perform the join using a technique that is a modified form of the plane-sweep method [160].

In the MSJ method, objects are assigned to levels of regular grids. Each level of grid cells is derived by dividing each dimension of the previous level in two, as shown in Figure 4.3. The first level, level zero, encloses the entire data space. An object is assigned to the finest (highest) level in which it does not cross a grid boundary, thereby creating an *MX-CIF quadtree index* [107]. Each part of Figure 4.3 shows an object assigned to a different level. Since each dimension is split, the fan-out will be large for higher dimensions, thereby limiting the depth of grid levels since an enormous number of lower-level cells would create an unwieldy index.

By assigning objects to cells in this manner, some small objects will be assigned to shallower (coarser) levels, as shown in Figure 4.3a. To overcome this inefficiency, Dittrich and Seeger introduce redundancy, thereby allowing an object to be replicated into multiple objects that can be assigned to a more appropriate level. For example, object a in Figure 4.3a would be replicated into four objects, each assigned to one of the four middle grid cells of level 2. This is an *expanded MX-CIF quadtree index* [3]. Unfortunately, in higher dimensions, an object could cross boundaries in many dimensions and would need to be replicated into 2^k cells for k boundary crossings. For this reason, objects are not replicated if k exceeds a predefined limit. Dittrich and Seeger [51] experimentally found that a small k factor, such as two or three, is best. By introducing redundancy in the dataset, duplicate results might

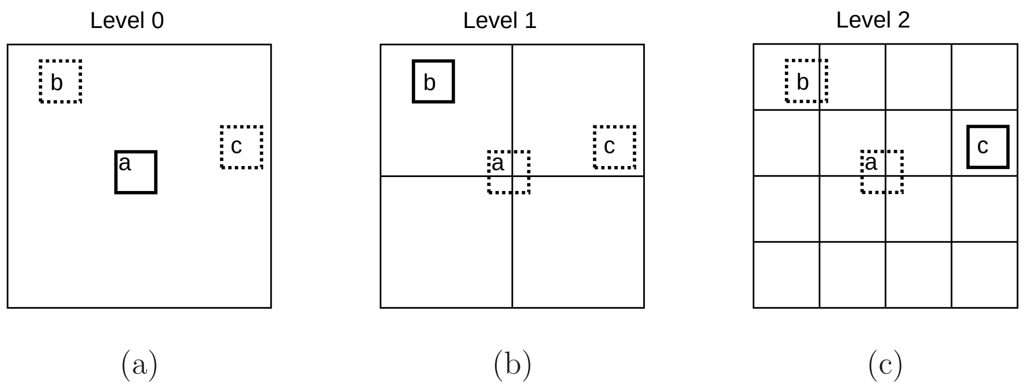


Figure 4.3: In the MSJ and GESS methods, objects are created out of multi-dimensional points by creating hyper-squares centered around each point with ϵ length sides. Each object is assigned to the finest regular grid in which it does not cross a grid boundary. In this example, only the square with the solid border in each figure is assigned to that level, with level 2 being the lowest level. (a) Object a is assigned to the root space, level 0, since it intersects the first (coarsest) grid level with four grid cells, that is, level 1. (b) Object b is assigned to the next level since it intersects the next finer grid level with sixteen cells, that is, level 2. (c) Object c is assigned to the lowest level.

appear, which can be removed in-line using a technique described later.

Once the data is assigned to grid cells, it is sorted based on the assigned grid cell using a linear order such as a Hilbert or Peano-Hilbert order [99, 170]. The data is secondarily sorted by level so that objects assigned to larger grid cells are seen first, immediately followed by objects within the larger grid cells that have been assigned to the next lower level of grid cells, and so forth.

After the data has been sorted, it is scanned in sorted order and joined using a technique attributable to Orenstein [143] and similar to a plane-sweep technique [160]. As each object is encountered, it is pushed onto a stack and joined with all other data on the stack. Once all of the data within a grid cell has been encountered, that data can be purged from the stack since it could not possibly be similar to any data yet to be seen because of the sorted order.

Because of the replication, duplicate results could be reported. Dittrich and Seeger use a *reference point method* [14, 50] to prevent duplicate results from being reported. In this method, a consistently chosen corner of the hyper-dimensional rectangle formed by the intersection of two ϵ expanded points is used, such as the lower left corner of a two-dimensional intersection. If this corner point is within the grid cell to which the objects are assigned, then the result (similarity) is reported. Since this point can only lie within one grid cell, duplicate results are avoided.

Nevertheless, MSJ and GESS methods are essentially grid-based indexing techniques because the points are assigned to a uniform grid in order to apply a linear order, and the methods have the drawbacks that we have discussed, thereby making them unsuitable for high-dimensional similarity joins. In fact, Dittrich and Seeger acknowledge this [51] when they state that "deficiencies of this method for high-dimensional intersection joins are that a high fraction of the input relation will be in level 0. The hypercubes in level zero [32], however, need to be tested against the entire input relation in a nested-loop manner." Analytically, using equation 3 from their paper [51], it can be easily confirmed that a large percentage of objects will reside at level zero. This poor performance is confirmed by the results of experiments as reported in Section 4.4.

4.2 The Quickjoin Algorithm

This section describes the details of the Quickjoin algorithm, as well as variations of the algorithms. Also, slightly different algorithms are used depending on whether the entire dataset fits into internal memory. The internal memory method using ball partitioning is presented in Section 4.2.1, and the generalized hyperplane variant of the Quickjoin algorithm is described in Section 4.2.2. A variant of the algorithm that assumes vector data is described in Section 4.2.3. Next, the external memory algorithm is described in Section 4.2.4, which uses the internal memory algorithm to join partitions of the data that are small enough to fit into internal memory. Finally, Section 4.2.5 describes a technique for improving the performance of the internal memory algorithm.

4.2.1 Quickjoin Using Ball Partitioning

The internal memory version of the Quickjoin algorithm, shown in Figure 4.4, takes three arguments: the desired ϵ, `eps`; a distance function, `distFunc`; and the set of objects to be joined, `objs`, which is implemented as an array for the best performance for the internal memory algorithm. The procedure recursively partitions the data in `objs` into smaller subsets until the subsets are small enough (defined by the constant `constSmallNumber`) to be efficiently handled by a nested-loop join, which is invoked on line 4 of the algorithm. Shown in Figure 4.5, the `NestedLoop` procedure simply performs a nested-loop join on the objects and reports the pairs of objects within `eps` of each other.

```
procedure Quickjoin(eps, distFunc, objs)
begin
  if objs.size < constSmallNumber then
    NestedLoop(eps, distFunc, objs);
    return;
  endif;

  p1←randomObject(objs);
  p2←randomObject(objs-p1);
  (partL, partG, winL, winG)←
        Partition(eps, distFunc, objs, p1, p2);

  QuickjoinWin(eps, distFunc, winL, winG);
  Quickjoin(eps, distFunc, partL);
  Quickjoin(eps, distFunc, partG);
end
```

Figure 4.4: The Quickjoin algorithm recursively partitions the data into subsets that are small enough to be efficiently processed by a nested-loop join. The "windows" around the partition boundary, `winL` and `winG`, are handled by a separate recursive function, QuickjoinWin, shown in Figure 4.8.

```
procedure NestedLoop(eps, distFunc, objs)
begin
  foreach a ∈ objs do
    foreach b ∈ objs do
      if a ≠ b AND distFunc(a,b)≤eps then
        Report(a,b);
      endif;
    enddo;
  enddo;
end
```

Figure 4.5: The NestedLoop procedure performs a self join on the set of objects, `objs`, finding all similar pairs of objects.

If there are more than a few objects, the set of objects is partitioned in two using the `Partition` function on lines 10 and 11. The `Partition` function, shown in Figure 4.6, requires two random objects, `p1` and `p2`, which are chosen on lines 8 and 9, of the Quickjoin algorithm. The `Partition` function divides the objects into two sets, `partL` and `partG`, and also creates two “window” subsets, `winL` and `winG`, that contain objects that are within `eps` of the opposing partition, that is, `winL` contains objects in `partL` that might be within `eps` of the objects in `partG`, and `winG` contains objects in `partG` that might be within `eps` of objects in `partL`. The window partitions are created, since, as was shown in Figure 4.1a, a simple partitioning in two will miss reporting some pairs of objects that are similar because they lie in different partitions. For the internal memory algorithm, the window partitions can use references to the objects rather than duplicate the objects to save internal memory and replication costs.

Lines 14 and 15 recursively reinvoke the `Quickjoin` procedure to find similar objects within the `partL` and `partG` partitions. To find similar objects between the two partitions, that is, similar pairs of objects containing one object from `partL` and one object from `partG`, the two window sets, `winL` and `winG` are joined. This is done using a modified version of the `Quickjoin` procedure, termed `QuickjoinWin`, shown in Figure 4.8, which is invoked on line 13.

The key to the Quickjoin algorithm is the `Partition` function, given in Figure 4.6, which partitions the data using the distance of every object to a random object (the pivot), `p1`, as shown in Figure 4.7. The variable `r` is the *pivot distance*, calculated on line 4, which, in this instance, is the distance from `p1` to a second random object, p2, that is used to partition the objects. Objects with a distance from p1 less than or equal to r form the partition partL, and objects with a distance from p1 greater than r form the partition partG.

The Partition function uses in-place partitioning, which assumes that the objects are stored in an internal memory array. The array is partitioned in place, which results in placing objects whose distance from p1 is greater than r to the right of all objects whose distance from p1 is less than or equal to r. The outer loop, starting at line 9, scans the objects from both ends of the array using two inner loops: one from the end (right side) on line 10, and one from the start (left side) on line 14. When out of place objects are found on each side, the two objects are exchanged on line 24. Additionally, as the objects are scanned, they are placed into window partitions if they are within eps of r. This is done on lines 11, 15, 21, 22, 34, and 37. Finally, the object dividing the two sets is checked starting on line 31, and serves as the dividing point for returning the two ends of the array as partitions, on line 43, along with the window partitions.

Different partitioning methods can also be used in the Partition function. The data can be partitioned using one random object and aggregate properties of the distances. For instance, the data can be partitioned based on half the distance to the farthest object. The data could also be partitioned based on the average distance or the median distance in order to ensure a more equal-sized partitioned. Methods other than ball partitioning are described in Section 4.2.2 and Section 4.2.3.

The QuickjoinWin procedure, shown in Figure 4.8, is only slightly different than the Quickjoin procedure. A nested-loop join is also used on line 5, which is

```
function Partition(eps, distFunc, objs[], p1, p2)
          : (partL, partG, winL, winG)
begin
   r←distFunc(p1,p2);
   startIdx←0;
   endIdx←objs.length-1;
   startDist←distFunc(objs[startIdx], p1);
   endDist←distFunc(objs[endIdx], p1);
   while startIdx < endIdx  do
      while endDist > r AND startIdx < endIdx do
         if endDist <= r+eps then winG.insert(objs[endIdx]);
         endDist←distFunc(objs[--endIdx], p1);
      enddo;
      while startDist <= r AND startIdx < endIdx do
         if startDist >= r-eps) winL.insert(objs[startIdx] then
         startDist←distFunc(objs[++startIdx], p1);
      enddo;

      if startIdx < endIdx then  // exchange objects
         // check if either goes in a window
         if endDist >= r-eps then winL.insert(objs[endIdx]);
         if startDist <= r+eps then winG.insert(objs[startIdx]);
         // exchange items
         objs[startIdx]↔objs[endIdx];
         startDist←distFunc(objs[++startIdx], p1);
         endDist← distFunc(objs[--endIdx], p1);
      endif;
   enddo;

   // clean up last object
   if startIdx == endIdx then
         // check if this needs to be added to a window
         if endDist > r AND endDist <= r+eps then
            winG.insert(objs[endIdx]);
         endif;
         if startDist <= r AND startDist >= r-eps then
            winL.insert(objs[startIdx]);
         endif;
         // check last object
         if endDist > r then endIdx--;
    endif;

    return(objs[0:endIdx], objs[endIdx+1:objs.length-1], winL, winG);
end
```

Figure 4.6: The `Partition` function partitions the data into two sets, those whose distances from object `p1` are greater than the partitioning distance, `r`, and those whose distances from `p1` are less than or equal to `r`. The value `r` is the distance from the randomly chosen object `p1` to another randomly chosen object, `p2`. Also, objects are inserted into two other "window" sets if the objects are within ϵ, `eps`, of `r`.

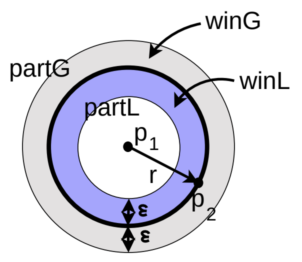


Figure 4.7: Objects are partitioned into two partitions – `partG` contains objects whose distances from `p1` are greater than `r` and `partL` contains objects whose distances from `p1` are less than or equal to `r`. The objects are also replicated into two windows. The window, `winG` is a subset of `partG` containing objects within ϵ of `r` and `winL` is a subset of `partL` also containing objects within ϵ of `r`.

termed `NestedLoop2`, shown in Figure 4.9, in which only similar pairs with an object from each set are reported. Additionally, the `Partition` function is called separately on each set of objects on lines 12 through 15, partitioning both sets using the same random objects, `p1` and `p2`. In our notation, the `Partition` function partitions `objs1` into objects whose distances from `p1` are less than or equal to `r`, `partL1`, and objects whose distances from `p1` are greater than `r`, `partG1`. Similarly, `objs2` is partitioned into `partL2` and `partG2`. The two sets whose distances from `p1` are less than or equal to `r`, `partL1` and `partL2`, are recursively joined using `QuickjoinWin` on line 17, since the purpose of the `QuickjoinWin` procedure is to find pairs between the two object sets. Similarly, `partG1` and `partG2` are recursively joined on line 18. In addition, the partitioning creates windows of objects that are within ϵ of `r`. The set `winL1` is a subset of `partL1` that is within ϵ of `r` and `winG1` is a subset of `partG1` within ϵ of `r`. The sets `winL2` and `winG2` are defined similarly. To determine which

```
 1 procedure QuickjoinWin(eps, distFunc, objs1, objs2)
 2 begin
 3   totalLen←objs1.size+objs2.size
 4   if totalLen < constSmallNumber2 then
 5      NestedLoop2(eps, distFunc, objs1, objs2);
 6      return;
 7   endif;
 8
 9   allObjects←objs1 ∪ objs2;
10   p1←randomObject(allObjects);
11   p2←randomObject(allObjects-p1);
12   (partL1, partG1, winL1, winG1)←
13         Partition(eps, distFunc, objs1, p1, p2);
14   (partL2, partG2, winL2, winG2)←
15         Partition(eps, distFunc, objs2, p1, p2);
16
17   QuickjoinWin(eps, distFunc, partL1, partL2);
18   QuickjoinWin(eps, distFunc, partG1, partG2);
19   QuickjoinWin(eps, distFunc, winL1, winG2);
20   QuickjoinWin(eps, distFunc, winG1, winL2);
21 end
```

Figure 4.8: The `QuickjoinWin` procedure, which is similar to the `Quickjoin` procedure given in Figure 4.4, joins two sets of objects by partitioning the two sets and then recursively joining the partitions. Only result pairs with one object from each set are reported.

windows to recursively join, the only ones to consider are those between subsets of `objs1` and `objs2`. The join between `winL1` and `winL2` is covered by the join between `partL1` and `partL2`. Similarly, the join between `winG1` and `winG2` is covered by the join between `partG1` and `partG2`. The remaining combinations are performed – between `winL1` and `winG2` on line 19, and between `winG1` and `winL2` on line 20. See Section 4.2.2 for a graphical depiction of the `QuickjoinWin` partitioning in which hyperplane partitioning is used instead of ball partitioning.

```
procedure NestedLoop2(eps, distFunc, objsA, objsB)
begin
  foreach a ∈ objsA do
     foreach b ∈ objsB do
        if distFunc(a,b) ≤ eps then
           Report(a,b);
        endif;
     enddo;
  enddo;
end
```

Figure 4.9: The NestedLoop2 procedure performs a join on the two given sets objects, finding all pairs of similar objects between the two sets of objects, that is, pairs with an object from each set.

4.2.2 Quickjoin Using Generalized Hyperplane Partitioning

Instead of using ball partitioning, we could also use generalized hyperplane partitioning. In this case, the partitioning hyperplane is used to separate the objects on the basis of which of the two pivot objects is closer. Unfortunately, in a general metric space, the partitioning hyperplane does not exist explicitly. Instead, it is implicit and known as a *generalized hyperplane*. Therefore, we cannot, in general, compute the exact distance from an object a to the partitioning hyperplane. However, we do have a lower bound for it (see Lemma 4.4 in [85, p. 539]) which is given in Equation 4.1, where $dist(a, p1)$ is the distance from object a to random object *p1* and $dist(a, p2)$ is the distance from a to the second random object, *p2*.

$$dist = (dist(a, p1) - dist(a, p2))/2 \tag{4.1}$$

To implement this variation, the `Partition` function in Figure 4.6 is modified slightly. In particular, the only change is in how the distance to each object is calculated and the value of `r`, which becomes zero since objects closer to object `p1`

```
(1) r←0;
(2) dist←(distFunc(a,p1)-distFunc(a,p2))/2;
```

Figure 4.10: (1) The pivot distance is zero for hyperplane partitioning and replaces the code on line 4 of Figure 4.6. (2) The distance function used for generalized hyperplane partitioning. This code replaces the distance function on lines 7, 8, 12, 16, 25, and 26 in Figure 4.6.

will have a positive distance and objects closer to `p2` will have a negative distance. These changes are show in Figure 4.10.

Equation 4.1 works for any distance metric. If exact distances from the partitioning hyperplane are known, then the number of items in the windows can be reduced by using these exact distances instead of lower bounds on the distances, which improves the performance of the algorithm. For example, when the distance metric is the L_2-norm, the exact form of Equation 4.1 is given by Equation 4.2, where $dist(p1, p2)$ is the distance between the two random objects *p1* and *p2*.

$$dist = (dist(a, p1)^2 - dist(a, p2)^2)/(2 * dist(p1, p2)) \tag{4.2}$$

To use this version of the hyperplane, Equation 4.2 is used to calculate distance, replacing the distance function in Figure 4.6, as was done with the generalized hyperplane function in Figure 4.10. Figure 4.11 shows the use of the hyperplane for partitioning, while Figure 4.12 shows its use for partitioning within the `QuickjoinWin` procedure.

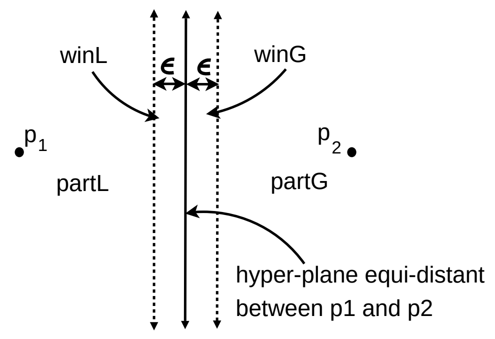


Figure 4.11: Using hyperplane partitioning, objects are partitioned into two sets – `partG` containing objects whose distances from the partitioning hyperplane are greater than zero, and `partL` containing objects whose distances to the partitioning hyperplane are less than or equal to zero. The objects are also inserted into two windows. The window, `winG` is a subset of `partG` containing objects within ϵ of the partitioning hyperplane, and `winL` is a subset of `partL` also containing objects within ϵ of the partitioning hyperplane.

4.2.3 Dimensional Version – Vector Data

If the data is represented as vectors, then another partitioning approach is to partition the data using one of the dimensions of the vector data. This partitioning function requires only slight modifications to the `Partition` function given in Figure 4.6, which are shown in Figure 4.13. In this case, each invocation of the `Partition` function must choose a dimension with which to partition the data. For instance, the first invocation could use the first dimension to partition the data, and subsequent calls could use the second dimension, then the third dimension, and so forth, cycling through the dimensions.

With this partitioning method, the distance to each object does not need to be calculated, but a randomly chosen object still needs to be chosen to partition the data. Alternatively, as with ball partitioning, a median or average value could

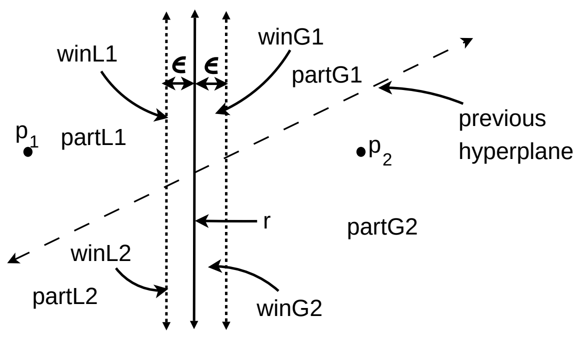


Figure 4.12: Objects are partitioned into four sets in the procedure `QuickjoinWin`, given in Figure 4.8. The procedure partitions sets `objs1` and `objs2`, which were divided by a previous hyperplane, as shown. Partitions `partG1` and `partG2` contain objects from object sets `objs1` and `objs2`, respectively, that are greater than the partitioning distance, `r`, and object sets `partL1` and `partL2` similarly contain objects less than or equal to the partitioning distance `r`, again from object sets `objs1` and `objs2`, respectively. As in the `Quickjoin` procedure, the objects are also replicated into windows containing objects within ϵ of the partitioning hyperplane.

be used. Note that even if the dataset is not represented by vectors, as long as we have a distance function, we can use an embedding method [62, 89, 121, 202] to transform it into a multi-dimensional vector space.

4.2.4 External Memory Algorithm Using Multiple Pivots (Partitions)

In order to improve I/O performance, we present an external memory algorithm which differs from the internal memory algorithm in that it uses multiple pivots during partitioning. Instead of creating two sets at a time, n partitions are created for each call to the `Partition` function, based on n randomly chosen objects (pivots). In this way, I/O is reduced since multiple partitionings occur at once and smaller partitions are created, thereby allowing for the objects to be read entirely

```
(1) d←pickPartitioningDimension();
(2) r←p1[d];
(3) dist←obj[d]
   (e.g., startDist←objs[startIdx][d];)
```

Figure 4.13: Modifications to the `Partition` function from Figure 4.6 to implement the dimensional variant of the Quickjoin method. (1) Between lines 3 and 4, a single dimension, `d` is chosen, which can be done in many ways, such as cycling through the dimensions. (2) The pivot, `r`, on line 4 becomes one dimension of the randomly chosen object `p1`. (3) The value used for dimensional partitioning. This code replaces the distance function on lines 7, 8, 12, 16, 25, and 26. As an example, line 7 of the `Partition` function is rewritten to use the `d`th dimension of object `objs[startIdx]`. The object `p2` is not needed.

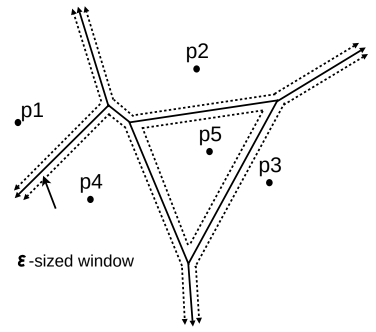


Figure 4.14: Multiple partitions are created using multiple pivots, $p1$ through $p5$, along with corresponding window partitions (dashed lines).

into internal memory sooner. A 2-dimensional example is shown in Figure 4.14 using five pivots, which results in partitions that form a Voronoi diagram [200]. This technique is similar to the multiple-pivot similarity search technique used by GNAT (Geometric Near-neighbor Access Tree) [33] for generalized hyperplane partitioning and an extension of the vp-tree [211] for ball partitioning.

The partitions formed by the external memory algorithm are stored externally on disk and can be read and written sequentially using block I/O for efficiency.

Once the partitions are small enough to fit in internal memory, the internal memory algorithm can be called for faster processing.

The external memory algorithm using multiple pivots is given in Figure 4.15. First, if there is sufficient internal memory, the externally stored objects, `objFile`, are read into memory and the internal memory algorithm is used on line 5. Otherwise, `NUM_PIVOTS` random objects are chosen on line 9, and the objects are partitioned on line 10 using the `PartitionExt` function, given in Figure 4.16. Note that since the objects are in external memory, the random objects (pivots) can be retrieved using non-sequential access, or, more efficiently, they can be selected and saved when the objects are written to external memory during the previous partitioning phase. The external partitioning function, `PartitionExt`, places each object into one `part[]` partition, corresponding to the closest pivot. The `win` set of partitions contains duplicates of the objects (stored externally) which correspond to objects that might be within `eps` of a partition. For instance, the `win[i][j]` partition correspond to sets of objects that are within the `part[i]`, but might be within `eps` of objects in `part[j]`. The `QuickjoinExt` function is recursively called on line 13 to join the `part` partitions while the `QuickjoinWinExt` procedure, shown in Figure 4.17, is called on line 17 to join the `win` partitions.

The `PartitionExt` function, shown in Figure 4.16, partitions the objects around a set pivots. The concept is similar to generalized hyperplane partitioning in Section 4.2.2, where a generalized hyperplane exists between each pair of objects. For each object, the nearest pivot is found using the loop on line 9, and the object is inserted into the corresponding `part` partition on line 16. Then, in

```
1  procedure QuickjoinExt(eps, distFunc, objFile)
2  begin
3    if sufficientInternalMemory(objFile.size) then
4        objs =readIntoInternalMemory( objFile );
5        Quickjoin( eps, distFunc, objs );
6        return;
7    endif;
8
9    pivots←selectRandomPivots( NUM_PIVOTS );
10   (part[], win[][])←PartitionExt(eps, distFunc, objFile, pivots);
11
12   for i=0:NUM_PIVOTS-1 do
13    QuickjoinExt(eps, distFunc, part[i]);
14   enddo;
15   for i=0:NUM_PIVOTS-2 do
16     for j=i+1:NUM_PIVOTS-1 do
17       QuickjoinWinExt(eps, distFunc, win[i][j], win[j][i] );
18     enddo
19   enddo
20 end
```

Figure 4.15: The external memory version of the Quickjoin algorithm.

the loop starting on line 18, the distance to every other pivot is checked and if the distance to the generalized hyperplane, calculated on line 21, is less than or equal to `eps`, the object is also added to the corresponding window, `win`, partition. As in Section 4.2.2, exact distances, if applicable, can be used to calculate the hyperplane on line 21 to improve performance. Note that since the data resides in external memory, the objects should be read sequentially in blocks for efficiency.

The `QuickjoinWinExt`, shown in Figure 4.17, joins two sets objects, `objFile1` and `objFile2`, using multiple pivots. Both sets of objects are partitioned using the same set of pivots on lines 12 and 13. Next, corresponding `part1` and `part2` partitions are joined on line 18 as well as corresponding window partitions on lines 22 and 23.

```
function PartitionExt(eps, distFunc, objFile, pivots)
         : (part[], win[][])
begin
   Initialize part and win files.

   while !EOF(objFile) do
      a←objFile.ReadObject();
      minDist←MAX_DOUBLE;
      foreach i=0:NUM_PIVOTS-1  do
         dist←distFunc(a,pivots[i]);
         if dist < minDist  then
            minDist←dist;
            minPivot←i;
         endif;
      enddo;
      part[minPivot].insert(a);
      p←pivots[minPivot];
      foreach i=0:NUM_PIVOTS-1  do
         if i != minPivot then
            pX←pivots[i];
            dist←distFunc(a,p)-distFunc(a,pX))/2;
            if -dist<eps then
               win[i][j].insert(a);
            endif;
         endif;
      enddo;
   return (part, win);
end
```

Figure 4.16: The external partitioning function determines, for each object `a`, which pivot, `p`, is the closest, and places the object in the corresponding `part` partition. Also, for every other pivot, `pX`, if the distance from `p` to the generalized hyperplane between `p` and `pX` is less than or equal to `eps`, then the object `a` is also inserted into the corresponding window partition.

```
procedure QuickjoinWinExt(eps, distFunc, objFile1, objFile2)
begin
  totalSize←objFile1.size + objFile2.size;
  if sufficientInternalMemory(totalSize) then
      objs1←readIntoInternalMemory(objFile1);
      objs2←readIntoInternalMemory(objFile2);
      QuickjoinWin(eps, distFunc, objs1, objs2);
      return;
  endif;

  pivots←selectRandomPivots(NUM_PIVOTS);
  (part1[], win1[][])←
        PartitionExt(eps, distFunc, objFile1, pivots);
  (part2[], win2[][])←
        PartitionExt(eps, distFunc, objFile2, pivots);

  for i=0:NUM_PIVOTS-1 do
   QuickjoinWinExt(eps, distFunc, part1[i], part2[i]);
  enddo;
  for i=0:NUM_PIVOTS-2 do
    for j=i+1:NUM_PIVOTS-1 do
      QuickjoinWinExt(eps, distFunc, win1[i][j], win2[j][i]);
      QuickjoinWinExt(eps, distFunc, win1[j][i], win2[j][i]);
    enddo;
  enddo;
end
```

Figure 4.17: The `QuickjoinWin` function from Figure 4.8 is also modified to create multiple partitions at a time.

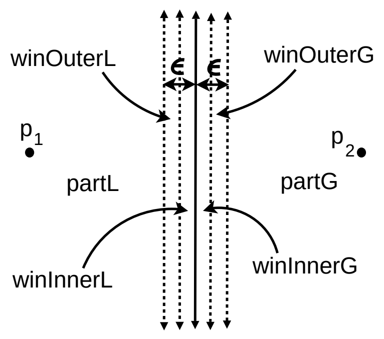


Figure 4.18: For internal partitioning, the windows can be split further in order to improve performance. As shown with generalized hyperplane partitioning, each window is split so that the inner windows, (`winInnerL` and `winInnerG`), contain objects within $\frac{\epsilon}{2}$ of the hyperplane, while the outer windows (`winOuterL` and `winOuterG`) contain the remaining objects that lie within ϵ of the hyperplane. Since `winOuterL` and `winOuterG` are more than ϵ apart, they do not need to be joined.

4.2.5 Split Windows

As a performance improvement, when the data is partitioned in two for internal partitioning (see Figure 4.6), instead of two windows, four windows can be returned. This is illustrated in Figure 4.18 with generalized hyperplane partitioning (Section 4.2.2), although any form of partitioning could be used as well. In the figure, the window partition `winL` from Figure 4.11 is split into partitions `winInnerL` and `winOuterL`, and the window partition `winG` is split into `winInnerG` and `winOuterG` partitions. The inner windows (`winInnerL` and `winInnerG`) contain the window objects within $\frac{\epsilon}{2}$ of the pivot or hyperplane, and `winOuterL` and `winOuterG` contain the remaining objects. In this way, the join between the two outer windows, `winOuterL` and `winOuterG`, can be avoided since they are further than ϵ apart. The remaining combinations of split windows are joined, as shown in Figure 4.19.

```
procedure Quickjoin(eps, objs, distFunc)
begin
  if objs.size<constSmallNumber then
   NestedLoop(eps, distFunc, objs);
   return;
  endif;

  p1←randomObject(objs);
  p2←randomObject(objs-r);
  (partL, partG, winInnerL, winOuterL, winInnerG,
winOuterG)←
     PartitionSplitWindows(eps, distFunc, objs, p2, p2);

  QuickjoinWin(eps, distFunc, winInnerL, winInnerG);
  QuickjoinWin(eps, distFunc, winInnerL, winOuterG);
  QuickjoinWin(eps, distFunc, winOuterL, winInnerG);
  Quickjoin(eps, distFunc, partL);
  Quickjoin(eps, distFunc, partG);
end
```

Figure 4.19: A variant of the Quickjoin function given in Figure 4.4 modified to subdivide each window into two parts to improve performance, as illustrated in Figure 4.18

4.3 Analysis

The performance of the Quickjoin algorithm in terms of both CPU and I/O depends on how well the data is partitioned at each step. As with the Quicksort algorithm [86], because of the random nature of the algorithm, the Quickjoin algorithm will likely partition the data reasonably well on average, and each data object will be partitioned $O(\log(n))$ times. However, at each level of recursion, the size of the dataset grows due to the window partitions, leading, as will be shown in Section 4.3.1, to $O(n(1+w)^{\lceil \log(n) \rceil})$ performance on average, where w is the average fractional size of the window partitions. As shown experimentally in Section 4.4.1, w is dependent upon ϵ, and as expected, the algorithm will perform well for smaller ϵ, and performance will degrade for larger ϵ values, as do all algorithms. Portions of the in-depth, supporting analysis are described separately, in Sections 4.3.3 and 4.3.4.

Section 4.3.2 discusses the I/O performance of the external memory algorithm, and for internal memory, the requirements of the algorithm are quite low. In the algorithm, a block of data, or even a single datum, can be read, and then immediately written to a partition. More internal memory will improve the performance by allowing larger partitions to be read into memory and to be fully processed, rather than having to incur the I/O cost of writing the data back out to external memory.

4.3.1 CPU Performance Analysis

The basic performance analysis for the `Quickjoin` procedure from Figure 4.4 is given in Equation 4.3, where QJ denotes a call to the `Quickjoin` procedure, P denotes a call to the `Partition` function from Figure 4.6, and QJW denotes a call to the `QuickjoinWin` procedure from Figure 4.8. Variable x represents the size of the object set. The `Partition` function divides the x objects into two partitions, represented by ax and $(1-a)x$, where a is the fractional size of the first partition and $(1-a)$ is the fractional size of the second partition. These two partitions are passed to the `Quickjoin` procedure, as represented by $QJ(ax)$ and $QJ((1-a)x)$. The `Partition` function also creates two window partitions, which are passed to the `QuickjoinWin` procedure, as represented by $QJW(wax, w(1-a)x)$, where w represents the fraction of objects from each partition that lie within ϵ of the partition boundary (the generalized hyperplane, for example) and are inserted into the window partitions.

$$QJ(x) = P(x) + QJ(ax) + QJ((1-a)x) + QJW(wax, w(1-a)x) \tag{4.3}$$

The performance of the `QuickjoinWin` procedure is similar to the `Quickjoin` procedure, and is given in Equation 4.4, where x_1 and x_2 are the sizes of the two input partitions.

$$\begin{aligned} QJW(x_1, x_2) &= P(x_1) + P(x_2) + QJW(ax_1, ax_2) + QJW((1-a)x_1, (1-a)x_2) \\ &+ QJW(wax_1, w(1-a)ax_2) + QJW(w(1-a)x_1, wax_2) \end{aligned} \tag{4.4}$$

The partitioning function, P, scans its data, and applies the distance function once or twice to each object, depending on whether ball partitioning or hyperplane partitioning is used, respectively. Letting d be the cost of the distance function, the cost of partitioning the data is $P(x) = cdx$, where c is 1 or 2, plus any constant overhead. Using this equation to replace $P(x)$ and expanding Equation 4.3 using both Equations 4.3 and 4.4 yields Equation 4.5.

$$\begin{aligned} QJ(x) &= cdx + acdx + QJ(a^2x) + QJ(a(1-a)x) + QJW(wa^2x, w(a(1-a)x) \\ &+ (1-a)cdx + QJ((1-a)ax) + QJ((1-a)^2x) \\ &+ QJW(w(1-a)ax, w(1-a)^2x) + wacdx + w(1-a)cdx \\ &+ QJW(wa^2x, wa(1-a)x) + QJW(wa(1-a)x, w(1-a)^2x) \\ &+ QJW(w^2a^2x, w^2(1-a)^2x) + QJW(w^2a(1-a)x, w^2a(1-a)x) \end{aligned} \tag{4.5}$$

Collecting just the cdx terms and ignoring the QJ and QJW terms yields $cdx + cdx(1 + w)$, since the a terms drop out. Intuitively, in the algorithm, at each level of the recursion, all of the objects are partitioned once. Additionally, objects in each window are also partitioned, yielding the $(1 + w)$ term. Continuing the expansion further yields the sequence in Equation 4.6, where each level is shown in brackets.

$$QJ(x) = cdx+[cdx(1+w)]+[cdx(1+2w+w^2)]+[cdx(1+3w+3w^2+w^3)]+... \tag{4.6}$$

Equation 4.6 is the expansion of the recurrence shown in Equation 4.7. Intuitively, at each level, the number of objects partitioned is $(1+w)$ times the previous level.

$$QJ(x) = cdx \sum_{i=0}^{\infty} (1 + w)^i \tag{4.7}$$

This equation is unbounded and growing. However, the depth of recursion is limited. The Quickjoin algorithm partitions the data in the same manner as the Quicksort algorithm, which is known to have an average case depth of $O(\log(n))$ [101] (see Section 4.3.3 for an analysis). Using this fact yields Equation 4.8.

$$QJ(x) = cdx \sum_{i=0}^{\lceil \log(n) \rceil} (1 + w)^i \tag{4.8}$$

Using induction, it can be show that the summation in Equation 4.8 has an upper bound of $O((1 + w)^{\lceil \log(n) \rceil})$ (see Section 4.3.4 for proof), which yields a final bound on the Quickjoin algorithm of $O(n(1 + w)^{\lceil \log(n) \rceil})$. This bound shows that the Quickjoin algorithm will perform well ($O(n)$) for smaller average windows since the $(1+w)^{\lceil \log(n) \rceil}$ term will be small. This term will be maximum when w is one, meaning that all objects are put into the windows. In this case, $2^{\log_b(n)} \leq n$ for $b \geq 2$, where b is the constant base of the logarithm. In other words, the performance for the largest window size is $O(n^2)$. The theoretical worst case, as with the Quicksort algorithm, occurs if the depth of recursion approaches n. In this case, the Quickjoin algorithm would have performance $O(n2^n)$. However, this is unlikely to occur, and the algorithm can be written to avoid such worst cases by, for example, using median values instead of random objects.

Furthermore, the w factor is dependent on ϵ, and as shown experimentally in Section 4.4.1, this dependence is nearly linear. For smaller ϵ values, w is also small and will not have a significant impact on performance. In particular, only for larger ϵ values will w have a significant impact on performance.

4.3.2 I/O Analysis

For the external memory algorithm (Section 4.2.4), given a fixed amount of internal memory, the external data must be partitioned until each external partition will fit within internal memory, at which time the internal memory algorithm can be used (Section 4.2.1). The analysis to determine the amount of I/O is nearly identical to that of the performance analysis given in Section 4.3.1 since each object is read from memory once each time it is partitioned, but the I/O analysis differs in two aspects. First, since multiple pivots are used, the partition function calculates the distance function for each pivot. However, the previous analysis used $P(x) = cdx$ for the performance of the partitioning function and the additional pivots can be incorporated into the c factor. Furthermore, the multiple pivots create a larger fan-out which shortens the depth of the recursion. This would be incorporated into the base of the logarithm of Equation 4.8, leaving the analysis unchanged.

The second difference is that the external algorithm stops, switching to the internal algorithm, when there is sufficient internal memory to hold the partition. Letting M be the number of objects that will fit in internal memory, the depth of the recursion is $\log(n/M)$. Additionally, since the data is read in blocks, the n variable should be represented by N, where $N = \frac{n}{m}$, and m is the number of objects per block. Applying these two modifications to the final average case performance yields an I/O average performance of $O(N(1 + w)^{\lceil \log(N/M) \rceil})$.

4.3.3 Depth of Recursion

The Quickjoin algorithm partitions the data in the same manner as the Quicksort algorithm, which is known to have an average case depth of $O(\log(n))$ [101]. Here, we apply the average case analysis to the Quickjoin algorithm using Equation 4.3 from Section 4.3.1. Since we are interested in the depth of the recursion of the `Quickjoin` procedure from Figure 4.4, we can drop the call to the `QuickjoinWin` procedure, $QJW()$, yielding Equation 4.9, with $P(x)$ replaced by cx, as was done with Equation 4.5, but dropping the unnecessary d constant and letting n denote the total number of objects.

$$QJ(n) = cn + QJ(an) + QJ((1-a)n) \tag{4.9}$$

In the average case, Equation 4.9 becomes Equation 4.10, which takes the average of all possible partitionings.

$$QJ(n) = \left(\frac{1}{n}\left(\sum_{i=0}^{n-1}(QJ(i) + QJ(n-i)\right)\right) + cn \tag{4.10}$$

With a base case shown in Equation 4.11.

$$QJ(1) = \Theta(1) \tag{4.11}$$

Combining $QJ(i)$ and $QJ(n-i)$ and multiplying Equation 4.10 by n starts the following series of algebraic manipulations.

$$\begin{aligned}
nQJ(n) &= 2\left(\sum_{i=0}^{n-1} QJ(i)\right) + cn^2 \\
nQJ(n) - (n-1)QJ(n-1) &= 2QJ(n-1) + cn^2 - c(n^2 - 2n + 1) \\
nQJ(n) &= (n+1)QJ(n-1) + 2cn - c \\
\frac{QJ(n)}{n+1} &= \frac{QJ(n-1)}{n} + \frac{2c}{n+1} - \frac{c}{n(n+1)} \qquad (4.12)
\end{aligned}$$

Expanding Equation 4.12 and dropping the $\frac{c}{n(n+1)}$ term since it is immaterial to an upper bound calculation, leads to the following sequence of equations.

$$\begin{aligned}
\frac{QJ(n)}{n+1} &= \frac{QJ(n-1)}{n} + \frac{2c}{n+1} \\
\frac{QJ(n-1)}{n} &= \frac{QJ(n-2)}{n-1} + \frac{2c}{n} \\
\frac{QJ(n-2)}{n-1} &= \frac{QJ(n-3)}{n-2} + \frac{2c}{n-1} \\
&\dots \\
\frac{QJ(2)}{3} &= \frac{QJ(1)}{2} + \frac{2c}{3}
\end{aligned} \tag{4.13}$$

Summing these equations gives

$$\frac{QJ(n)}{n+1} = \sum_{i=3}^{n+1} \frac{2c}{i} = 2c\left(\log_e(n+1) + \gamma - \frac{3}{2}\right) \tag{4.14}$$

The final equation results from the fact that $\sum_{i=1}^{n+1} \frac{1}{i} = \log_e(n+1) + \gamma$, where the constant is $\gamma = 0.577$ (Euler's Constant). In other words, Equation 4.14 shows that the `Quickjoin` procedure will perform, on average, $O(n\log(n))$ calculations partitioning the data. Since, at each level of recursion, the `Quickjoin` function partitions all of the n data items, meaning that the `Quickjoin` function will terminate at $O(\log(n))$ depth, on average. Also, since the `QuickjoinWin` uses the same partitioning method, it will also terminate at $O(\log(n))$ depth.

4.3.4 Upper Bound On $\sum_{i=0}^{\lceil\log(n)\rceil}(1+w)^i$

$$QJ(x) = \sum_{i=0}^{\lceil\log(n)\rceil} (1+w)^i \tag{4.15}$$

We assert that Equation 4.15 has an upper bound of $O\left((1+w)^{\lceil \log(n) \rceil}\right)$. This is assertion is true if Equation 4.16 holds and $c \geq 1$, which we prove by induction.

$$\sum_{i=0}^{\lceil \log(n) \rceil} (1+w)^i \leq c(1+w)^{\lceil \log(n) \rceil} \tag{4.16}$$

First, the base case of $n = 1$ is as follows, where any $c \geq 1$ satisfies the relation:

$$\sum_{i=0}^{\lceil \log(1) \rceil} (1+w)^i = \sum_{i=0}^{0} (1+w)^i = (1+w)^0 = 1 \leq c(1+w)^{\lceil \log(1) \rceil} = c \tag{4.17}$$

Assuming Equation 4.16 is true for n, we prove the induction in two cases. First, if $\lceil \log(n) \rceil = \lceil \log(n+1) \rceil$, then the equation for $n+1$ becomes

$$\begin{aligned} \sum_{i=0}^{\lceil \log(n+1) \rceil} (1+w)^i &= \sum_{i=0}^{\lceil \log(n) \rceil} (1+w)^i \\ &\leq c(1+w)^{\lceil \log(n) \rceil} \\ &= c(1+w)^{\lceil \log(n+1) \rceil} \end{aligned} \tag{4.18}$$

Equation 4.18 holds for any $c \geq 1$. Next, if $\lceil \log(n) \rceil = \lceil \log(n+1) \rceil - 1$, then the equation for $n+1$ becomes

$$\begin{aligned}
\sum_{i=0}^{\lceil\log(n+1)\rceil} (1+w)^i &= (1+w)^{\lceil\log(n+1)\rceil} + \sum_{i=0}^{\lceil\log(n)\rceil} (1+w)^i \\
&\leq (1+w)^{\lceil\log(n+1)\rceil} + c(1+w)^{\lceil\log(n)\rceil} \\
&= (1+w)^{\lceil\log(n+1)\rceil} + \frac{c(1+w)^{\lceil\log(n)\rceil+1}}{1+w} \\
&= (1+w)^{\lceil\log(n+1)\rceil} + \frac{c(1+w)^{\lceil\log(n+1)\rceil}}{1+w} \\
&= (1+w)^{\lceil\log(n+1)\rceil} \cdot \left(1 + \frac{c}{1+w}\right) \\
&= c(1+w)^{\lceil\log(n+1)\rceil} \cdot \left(\frac{1}{c} + \frac{1}{1+w}\right) \\
&\leq c(1+w)^{\lceil\log(n+1)\rceil} \qquad (4.19)
\end{aligned}$$

The final equation, Equation 4.19, is true provided that

$$\frac{1}{c} + \frac{1}{1+w} \leq 1 \qquad (4.20)$$

Rewriting Equation 4.20 gives

$$c \geq \frac{1+w}{w} \qquad (4.21)$$

Since w is a positive constant, $\frac{1+w}{w}$ can be bounded by c, proving the bound of $O\left((1+w)^{\lceil\log(n)\rceil}\right)$.

4.4 Experiments

This section presents the results of similarity join experiments on real datasets. The algorithms were implemented in C++ and the experiments were performed on a computer with a Pentium 3 processor and 512MB of RAM, running Windows XP.

The similarity join algorithms were tested using color data from the UC Irvine repository [82]. The data consists of multi-dimensional features extracted from a Corel image data collection with 68,040 objects. The experiments presented here used the 9-dimensional color moment data, with data values ranging from -4.8 to 4.4, and 32-dimensional color histograms, with data values ranging from 0 to 1. For experiments with larger datasets, we randomly perturb the color data, creating data clustered around the original point set.

In the experiments, the variations of the Quickjoin algorithms are compared to the EGO algorithm, described in Section 4.1.1, and the GESS algorithm, described in Section 4.1.2. The EGO algorithm was implemented with the suggested EGO* improvements [103], and the block size of the EGO algorithm was tuned to maximize its performance. The GESS algorithm was implemented using both a Hilbert order and a Morton (Peano-Hilbert) order. Since both orders performed similarly, only results with the Hilbert order are shown. The GESS algorithm was also tuned to maximize performance by adjusting the k factor. The sort codes (the assigned grid cells) were limited to 32 bits, thereby limiting the maximum depth to one or two levels, which did not impact performance since in many of the experiments, most of the data was assigned to the first two levels (level 0 and 1), regardless.

As this section shows, the Quickjoin method generally performs much better than the EGO algorithm and in some cases performs significantly better, especially for the L_1-norm metric. However, the GESS method always performs significantly worse than the both the Quickjoin and the EGO methods. With the color data, the GESS method performs poorly because the data is skewed towards lower values, and most of the data is assigned to the same grid cell, resulting in a nested-loop join. Even with synthetic data in higher dimensions, most points have several features that are near the median values, which exceeds the GESS k factor, and most values are placed into level 0, resulting, again, in a nested-loop join. As quoted in Section 4.1.2, the creators of the GESS method anticipated these problems. The experiments in this section verify that the GESS method is not well-suited for high-dimensional similarity joins.

Experiments on datasets that only require internal memory (no disk I/O) are presented in Section 4.4.1, and experiments that require external memory (disk I/O) are presented in Section 4.4.2. While external memory is required to handle datasets of any size, the internal memory experiments not only demonstrate the various internal memory Quickjoin algorithms, but are also of practical significance since large datasets can be processed entirely in memory with today's internal memory limits.

4.4.1 Internal Memory Similarity Joins

This section presents the results of experiments using only internal memory and datasets that fit entirely into internal memory. The four internal memory variations of the Quickjoin algorithm were used in the experiments in this section:

1. The ball-partitioning method described in Section 4.2.1, referred to as QuickJoinBall in the plots.

2. The generalized hyperplane version described in Section 4.2.2, referred to as

QuickJoinGHP.

3. The L_2-norm hyperplane described in Section 4.2.2, referred to as QuickJoinHP2.

4. The dimensional version for vector data described in Section 4.2.3, referred to as QuickJoinDim.

Each method was also implemented with the split windows enhancement described in Section 4.2.5.

The plots in this section show the time to perform the similarity join, assuming an unprocessed dataset. The Quickjoin algorithms immediately execute the join, while the EGO and GESS methods preprocess the data by sorting the data, and in the case of the GESS method, assigning the data linear order codes. Except for the smallest of experiments, this preprocessing tends to be only a small fraction of the total processing time.

The preprocessing times for the EGO and GESS methods are shown in Figure 4.20. For both methods, the sort time increases with increasing ϵ. The EGO method uses ϵ to calculate the EGO grid cell, as described in Section 4.1.1. Since the grid cells are of size ϵ, as ϵ increases, there are fewer cells in which to place objects, thereby making it more likely that two objects will be equal in a single dimension. Therefore, on average, more dimensions will be needed to determine which object is greater. For the GESS method, as ϵ increases, the size of each hyper-square around each point increases. This increases processing time by requiring more replication since the larger hyper-squares are more likely to intersect grid boundaries.

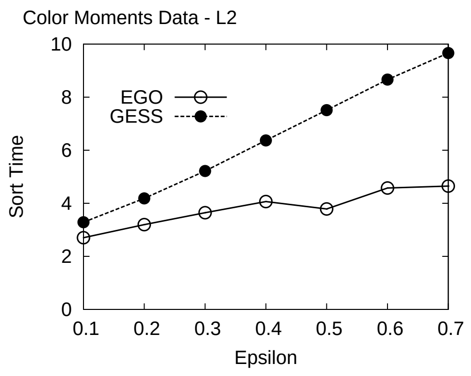
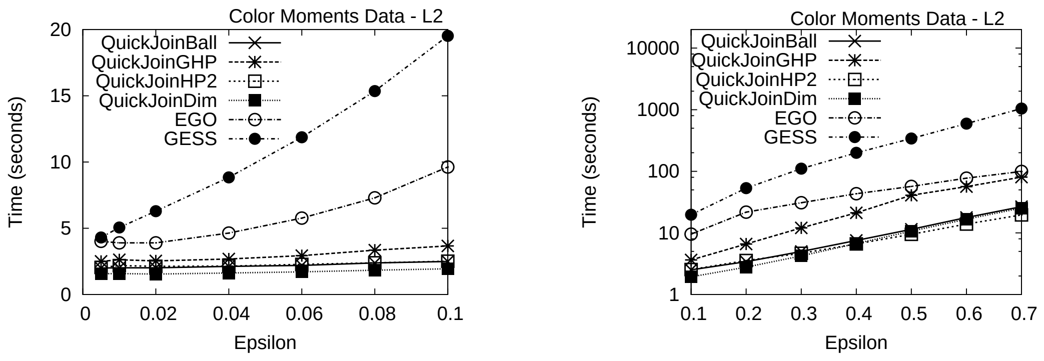


Figure 4.20: Sort time for the Color Moments data.

The results of similarity join experiments with the 9-dimensional color moment data, with data values ranging from -4.8 to 4.4, are shown in Figures 4.21a and 4.21b, both using an L_2-norm. Figure 4.21a shows experiments with smaller ϵ values (less than 1%), which represent queries where a few hundred results are reported, while Figure 4.21b shows experiments with larger ϵ values, which represent queries where more results are reported ranging from a few hundred up to several thousands. For smaller ϵ values, shown in Figure 4.21a, the Quickjoin algorithms outperform the EGO algorithm and significantly outperform the GESS algorithm. The Quickjoin algorithms, except for the generalized hyperplane version (QuickJoinGHP), exhibit an almost constant performance with increasing ϵ values. In these cases, the ϵ sized window partitions represent only a small fraction of the processing. For larger ϵ values, shown in Figure 4.21b, the Quickjoin algorithms begin to perform slightly worse. The ball partitioning, hyperplane partitioning and dimensional versions of the Quickjoin algorithm exhibit nearly identical performance on this dataset as well as most other datasets. The generalized hyperplane version (QuickJoinGHP) which

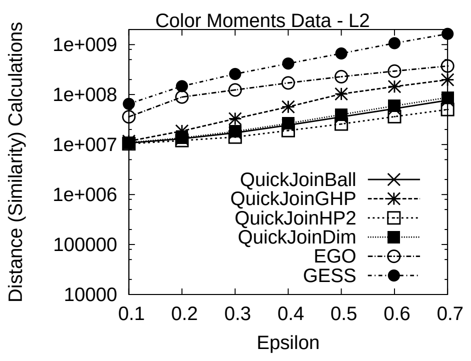


(a) (b)

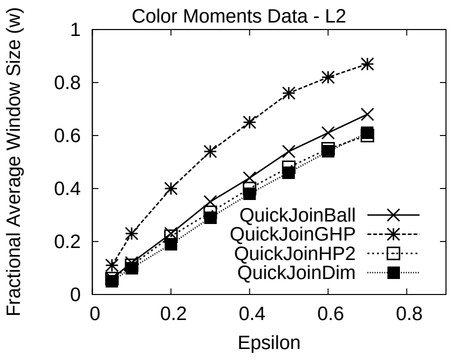
Figure 4.21: Experiments with color moment data for (a) smaller ϵ values using an L_2-norm, and (b) larger ϵ values using an L_2-norm.

makes use of a lower bound on the distance (hence resulting in a larger window) performs worse than these variations. Nevertheless, the Quickjoin algorithms still outperform both the EGO and the GESS algorithms.

Figure 4.22a shows the number of distance calculations for the experiments shown in Figure 4.21b, which demonstrates that the Quickjoin algorithms improved performance is due to the fewer distance (similarity) tests that it performs. Further insight into the performance of the Quickjoin algorithms can be gained by examining the fractional size of the windows created by the `Partition` function from Figure 4.6, which is called by the `Quickjoin` and `QuickjoinWin` procedures given in Figures 4.4 and 4.8. As shown in Figure 4.22b, the average window size increases nearly linearly with ϵ. Note that w is a function of ϵ. In this case, with the color moment data, the data can range in value from -4.4 to 4.8. In 9 dimensions, the maximum ϵ would be $\sqrt{9 \cdot (4.8 - (-4.4))^2} = 27.6$. However, w would be close to its maximum value of 1 at a smaller value, as can be seen with the plot of Quick-



(a)



(b)

Figure 4.22: For the color moment data using an L_2-norm, (a) the number of distance (similarity) calculations (note that the QuickJoinBall and QuickJoinDim algorithms performed identically, making it hard to see the plot of the QuickJoinBall performance), and (b) the average fractional window size created by the `Partition` function.

JoinGHP, where w approaches 1. Intuitively, the size of the result set increases dramatically for the ϵ values shown in the plot, and for even larger values, nearly all pairs of objects will be returned, resulting in $O(n^2)$ performance of the evaluated algorithms. Figure 4.22b also illustrates the fact that the generalized hyperplane version (QuickJoinGHP) uses window sizes that are twice as large as the L_2-norm hyperplane version (QuickJoinHP2).

Figure 4.23 shows the results of repeating the experiments in Figure 4.21b using an L_1-norm. In this case, the EGO and GESS algorithms are shown to perform nearly the same as before, while the performance of the Quickjoin method improves. Note that the QuickjoinHP2 method is not shown as it assumes the use of the L_2-norm. Because the distances between objects are larger when using the L_1-norm as compared to when using the L_2-norm, the windows created by the Quickjoin method are relatively smaller, thereby leading to improved performance. Since

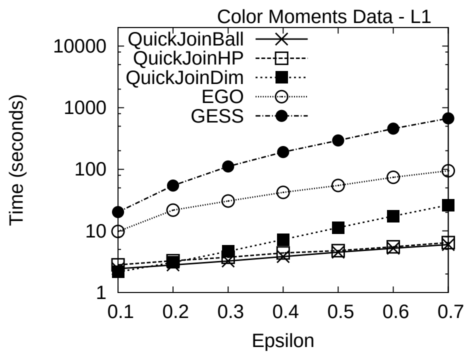


Figure 4.23: Experiments with color moment data for larger ϵ values using an L_1-norm.

the dimensional version of the Quickjoin algorithm (QuickjoinDim) does not use distance, it is unaffected by the change in the distance metric. Also, since the distances between objects is increased, the number of objects in the join result tends to be in the hundreds, rather than in the thousands, as with the join result from Figure 4.21b.

The results of experiments with the 32-dimensional color histogram data are shown in Figure 4.24. The data ranges in value from 0 to 1, and the ϵ values produce thousands of results. In this case, the Quickjoin algorithms significantly outperforms the EGO method. The GESS method is not shown since we limited the length of the linear order codes to 32 bits (the size of long integers with C++ on a Windows platform), and 32 dimensions plus the extra size bits exceeds this limit. The poor performance of the GESS method on previous experiments and some testing in higher dimensions did not warrant more extensive testing with the GESS method.

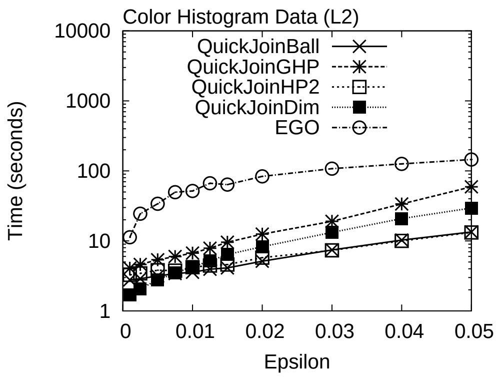


Figure 4.24: Experiments with 32-dimensional color histogram data, using the L_2-norm.

4.4.2 External Memory Similarity Joins

This section presents the results of experiments that require that the algorithms use external memory to perform the similarity join, which demonstrates the I/O performance of the algorithms. The overall performance of the Quickjoin algorithms is still better than that of the EGO and GESS methods, even though the Quickjoin methods perform slightly more I/O. Since the CPU costs are a significant factor in the performance of similarity joins due to the expensive distance (similarity) calculations, the Quickjoin methods perform better overall since their internal memory performance is so much better, as was shown in Section 4.4.1.

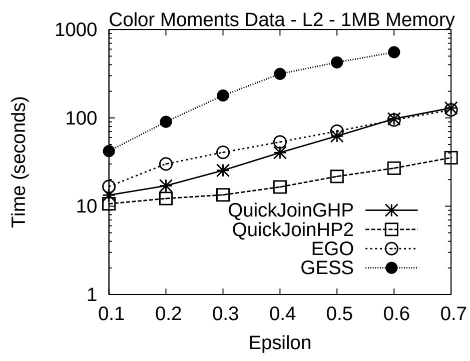
The algorithms were again tested using both sets of color data. This time restricting the amount of internal memory available to the algorithms, thereby forcing them to use external memory. The size of the color dataset was increased by duplicating points randomly and perturbing each feature of the point data by no more than 10%, resulting in data clustered around the original multi-dimensional point set.

The external memory version of the Quickjoin algorithm was implemented as described in Section 4.2.4. The window partitions are calculated using both the generalized hyperplane applicable to any distance metric (again termed QuickjoinGHP in the plots) and the L_2-norm hyperplane (again termed QuickjoinHP2 in the plots). As appropriate, the internal memory Quickjoin algorithm used is the corresponding generalized hyperplane or hyperplane version of the internal memory Quickjoin algorithm.

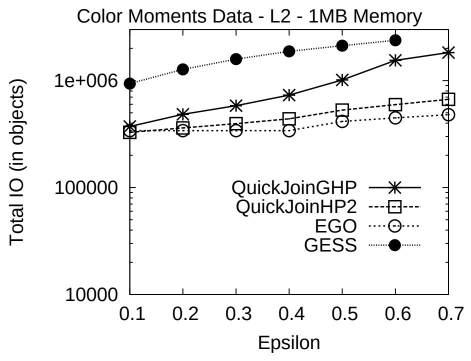
Figure 4.25a shows the results of experiments with external memory, which duplicate the experiment in Figure 4.21b, but with internal memory limits. The hyperplane version of the Quickjoin still performs better than the EGO method, while the performance of the generalized hyperplane version is similar to the EGO method. Figure 4.25b shows the I/O performance for the experiments in Figure 4.25a. Here we find that the hyperplane Quickjoin method performs slightly more input and output than the EGO method, but the generalized hyperplane version performs more I/O, which explains the performance difference shown in Figure 4.25a.

No results are given for the GESS method for an ϵ of 0.7 because the method was not able to process the data with the given amount of internal memory (1MB). Most of the data was inserted into the first level, which tends to stay in memory longer. In this instance, the GESS stack filled beyond the set limit. Since the creators of the MSJ and GESS method provide no contingency for such an occurrence, the method simply fails in this case.

Figure 4.26 shows the results of experiments with clustered color moments



(a)

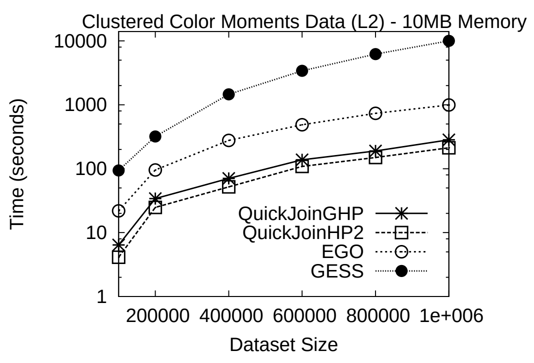


(b)

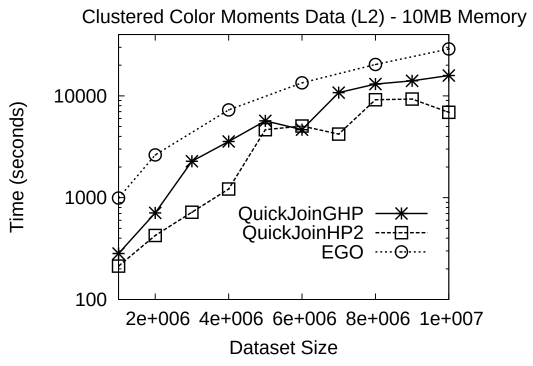
Figure 4.25: Experiments with color moment data for larger ϵ values using an L_2-norm and with memory restricted to 1MB. (a) Execution time, and (b) I/O.

data for very large datasets, which were created from the color moments data by duplicating the points and randomly perturbing each point, thereby creating clusters around each original point. In the experiments for up to a million objects, shown in Figure 4.26a, the Quickjoin algorithms are three to four times faster than the EGO method. An ϵ of 0.1 was used, with results sets ranging from four to five times larger than the datasets size. In other words, for a dataset of one million objects, there are approximately five million result pairs.

In experiments with up to ten million objects, shown in Figure 4.26b, the Quickjoin algorithms continue to perform two to three times better than the EGO method, although with less consistent results. The hyperplane algorithm (QuickJoinHP2) especially exhibits varying performance for larger data counts due to the random nature of the algorithm. Since the GESS method performed much worse for up to one million objects, it was not included in experiments for more than a million objects. As shown in Figure 4.27, for larger datasets, the difference in I/O



(a)



(b)

Figure 4.26: Experiments with clustered color moments data and internal memory restricted to 10MB for (a) up to one million points and (b) ten million points. An ϵ of 0.1 was used with an L_2-norm.

performance is negligible.

Figure 4.28 shows experiments from Section 4.4.1 repeated with the larger clustered datasets. The results of experiments with clustered 32-dimensional color histogram data, are shown in Figure 4.28a, using an ϵ of 0.05 and the L_2-norm. In this case, the Quickjoin method that uses the hyperplane tailored to the L_2-norm (QuickJoinHP2) performs much better than the generalized hyperplane Quickjoin method (QuickJoinGHP). The results of experiments with clustered 9-dimensional color moments data using an L_1-norm and an ϵ of 0.05 are shown in Figure 4.28b. The QuickJoinHP2 variation is not shown since it is tailored to the L_2-norm.

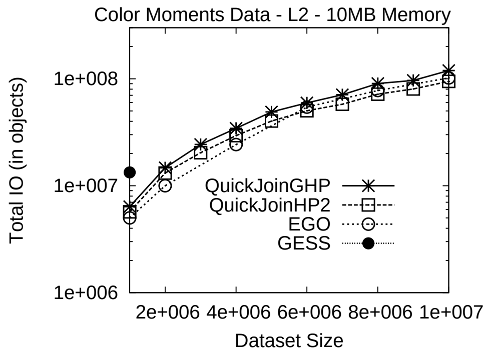


Figure 4.27: The I/O performance difference is negligible for larger datasets.

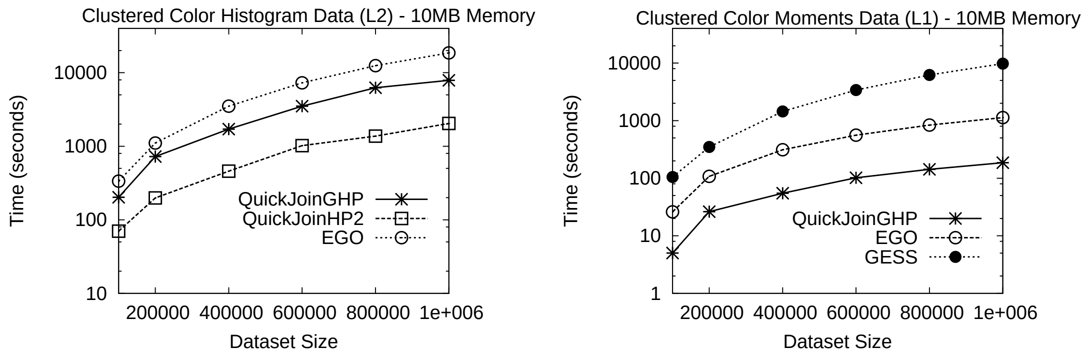


Figure 4.28: (a) Experiments with clustered 32-dimensional color histogram data, using the L_2-norm. (b) Experiments clustered with 9-dimensional color moments data, using the L_1-norm.

Chapter 5

An Incremental Hausdorff Distance Calculation Algorithm

In this chapter, we show that incremental distance join concepts [83, 179, 180, 206] can be used to improve the performance of the calculation of the Hausdorff distance for datasets that can be indexed by hierarchical containment spatial indices, such as the R-tree [76]. The incremental calculation of the Hausdorff distance requires that the incremental distance join algorithm be modified in two ways. The first modification is that rather than calculating the minimum distance between pairs of nodes and objects, an estimate of the Hausdorff distance between the pairs needs to be calculated. For the incremental distance join, the distance calculations that involve nodes actually provides a lower bound estimate of the minimum distance between objects contained within the nodes. This lower bound guarantees that if the head of the priority queue contains two objects, then they must be the closest pair since the lower bound on any other pair is greater. Because the Hausdorff distance is formulated as a maximum, the distance associated with pairs of elements in the priority queue is an upper bound. In addition, because the Hausdorff distance is the maximum of the minimum distance from set A to set B, the lower bound on the minimum distance between every object in set A to set B is also implicit in the calculation. This leads to the second modification to the incremental distance join which is that a second level of priority queues is maintained in order to estimate and find this minimum value. In particular, every element of the priority queue contains a node or object for the index on set A and a secondary priority queue of objects from set B, maintained in increasing order of the lower bound estimate on the minimum distance. Further details of the algorithm are presented in Section 5.3.

The rest of this chapter is organized as follows. After reviewing related work in Section 5.1, Section 5.2 describes the equations that are required to adapt the incremental distance join techniques to calculate the Hausdorff distance for bounding boxes. The modified algorithm is presented in Section 5.3, while Section 5.4 contains the results of experiments that show the improved performance.

5.1 Related Work

As we pointed out earlier, the Hausdorff distance can be computed by using a Voronoi diagram. If n is the size of set A and m is the size of set B, then, in two dimensions, the Hausdorff distance can be calculated in $O((n+m)\log(n+m))$ [8]. For image matching, a more robust use of the Hausdorff distance involves the calculation of the minimum Hausdorff distance under rotation and translation. Huttenlocher et al. [94] presented an algorithm to compute the Hausdorff distance under rigid motion transformation (rotation and translation) in two dimensions in time $O((m+n)^6 \log(nm))$. Due to the relatively high computational complexity of this algorithm, approximation methods have also been studied [7, 96]. In addition, partial matching techniques [95] calculate the k^{th} ranked Hausdorff distance, which can be used to mitigate the effects of noise by ignoring outliers.

The incremental distance join algorithm [83, 179, 180, 206] outputs the results of an all-pairs nearest-neighbor join in closest pair order. The algorithm assumes that the datasets are indexed with hierarchical containment indices, such as the R-tree [76]. The algorithm proceeds as follows. Initially, the pair of root nodes is placed on a priority queue, which maintains pairs of nodes or objects (or a combination of a node and an object) according to the distance between the pairs of nodes (objects), with the closest pair residing at the head of the priority queue. At each step, the head of the priority queue is examined and one or both nodes in the pair are expanded, as needed, and reinserted into the queue. Results are reported when the head of the priority queue contains two objects.

5.2 Hausdorff Distance Computations for Bounding Boxes

In order to apply the incremental join technique, the bounds on the Hausdorff distance between the objects in the hierarchical index must be calculated. To illustrate these concepts, we assume an R-tree like index structure [76], although a similar analysis could be applied to any hierarchical containment index for spatial data [11]. The two object types in the R-tree are the minimum bounding boxes (MBBs) of the index nodes and the data objects, such as points. The lower bound and upper bound on the distances between two data objects is the distance between the two data objects, where the distance between two objects, $dist(a, b)$, is defined by the user. In this section, the general case for the bounds between two MBBs is explained, which is illustrated in Figure 5.1, and the remaining combinations follow

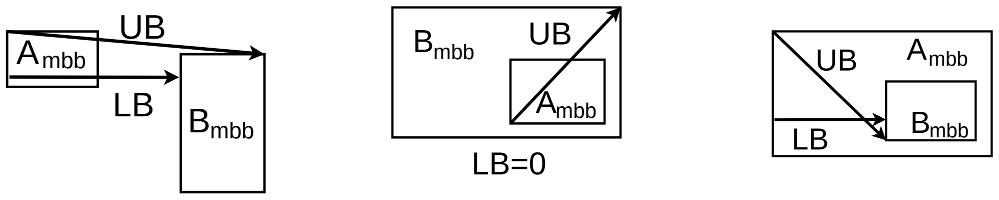


Figure 5.1: The lower bound (LB) and the upper bound (UB) on the directed Hausdorff distance between disjoint and overlapping minimum bounding boxes, A_{mbb} and B_{mbb}.

as degenerate cases.

The lower bound between two MBBs, shown in Equation 5.1, follows directly from the definition of the directed Hausdorff distance from Equation 1.1, and the fact that every face of the MBB must touch some object (else the MBB would be smaller).

$$LB\left[h(A_{mbb}, B_{mbb})\right] = \begin{cases} 0 & \text{if a within b,} \\ max_{f_a \in faces(A_{mbb})} min_{f_b \in faces(B_{mbb})} mindist(f_a, f_b) & \text{otherwise.} \end{cases} \quad (5.1)$$

Essentially, the bounds are derived by assuming that there is a single point somewhere within each face (edge in two dimensions), and then calculating the minimum and maximum values the Hausdorff distance could achieve. In this case, using $mindist$, the minimum distance between every face, $f_a \in faces(A_{mbb})$ and $f_b \in faces(B_{mbb})$, is calculated. Then, applying the definition of the directed Hausdorff distance, for each f_a, the minimum $mindist$ to each f_b is chosen, and the maximum of these values is the lower bound. As an exception, if object A_{mbb} is within object B_{mbb}, then the lower bound is zero. Equation 5.2 shows the upper bound equation, which is derived in the same manner as the lower bound, using the maximum distance, *maxdist*, between each face instead of *mindist*.

$$UB\left[h(A_{mbb}, B_{mbb})\right] = max_{f_a \in faces(A_{mbb})} min_{f_b \in faces(B_{mbb})} maxdist(f_a, f_b) \tag{5.2}$$

For the directed Hausdorff distance between an object, a, and an MBB, B_{mbb}, the lower bound is the minimum distance from a to B_{mbb}, as shown in Figure 5.2. If a is within B_{mbb} or on its border, also shown in Figure 5.2, then the lower bound on the directed Hausdorff distance is zero. The lower bound, LB, in equation form, is shown in Equation 5.3.

$$LB\left[h(a, B_{mbb})\right] = \begin{cases} 0 & \text{if a within b,} \\ \\ min_{f_b \in faces(B_{mbb})} mindist(a, f_b) & \text{otherwise.} \end{cases} \tag{5.3}$$

The upper bound, shown in Equation 5.4 and illustrated in Figure 5.2, is a degenerate case of Equation 5.2 and takes the minimum of the maximum distance from a to each face of B_{mbb} [1].

$$UB\left[h(a, B_{mbb})\right] = min_{f_b \in faces(B_{mbb})} maxdist(a, f_b) \tag{5.4}$$

For the directed Hausdorff distance between an MBB, A_{mbb}, and an object, b, the lower bound is the maximum of the minimum distance of each face of A_{mbb}

[1]Equation 5.2 is also referred to as the MinMaxDist distance [166] or the MaxNearestDist [171], and is used in nearest neighbor finding.

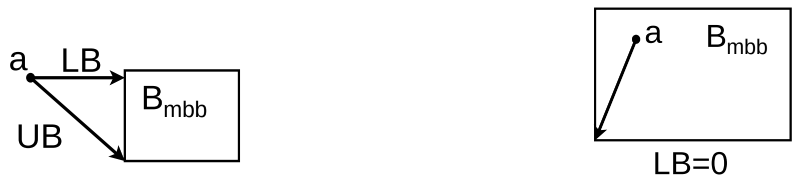


Figure 5.2: The lower and upper bound on the Hausdorff distance between an object (in this case, a point) a and an MBB, B_{mbb}.

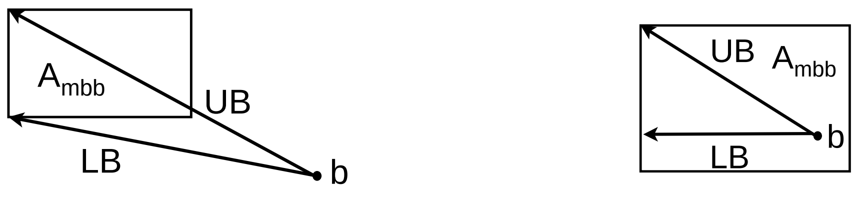


Figure 5.3: The lower and upper bound on the Hausdorff distance between an MBB, A_{mbb}, and an object b.

to b, as shown in Figure 5.3. The lower bound, LB, in equation form, is shown in Equation 5.5.

$$LB\left[h(A_{mbb}, b)\right] = max_{f_a \in faces(A_{mbb})} mindist(f_a, b) \tag{5.5}$$

Similarly, the upper bound on the directed Hausdorff distance between an MBB, A_{mbb}, and an object, b, is the maximum of the maximum distance between each face of A_{mbb} and b, as shown in Figure 5.3. The upper bound, UB, in equation form, is shown in Equation 5.6.

$$UB\left[h(A_{mbb}, b)\right] = max_{f_a \in faces(A_{mbb})} maxdist(f_a, b) \tag{5.6}$$

5.3 Algorithm Description

The Hausdorff incremental distance join algorithm calculates the directed Hausdorff distance from one set, A, to another set B, $h(A, B)$, as outlined in Section 1.3. Both sets are assumed to be indexed by a hierarchical containment index, such as an R-tree [76]. The key to the algorithm is the maintenance of two levels of priority queues, as shown in Figure 5.4. The main priority queue contains entries (index nodes and data objects) that correspond to the A index. Each entry, shown in Figure 5.4, consists of an element, `a`, from the A index, a secondary priority queue, `secondaryPQ`, that corresponds to elements in the B index, and the minimum upper bound on the directed Hausdorff distance, `qMinUB`, between `a` and elements, `b`, of `secondaryPQ`. The `qMinUB` value is found by calculating the upper bound as entries are inserted into `secondaryPQ`, which is maintained primarily to facilitate the calculation of `qMinUB` and to estimate the lower bound on the distance between element `a` and the B dataset. As noted later, the secondary queues also facilitate the calculation of partial Hausdorff distances. The main priority queue is sorted in decreasing order by `qMinUB`, which corresponds to an upper bound on the best guess of the directed Hausdorff distance for the `a` element. The secondary priority maintains its elements in increasing order by the lower bound on the distance between element `a` and element `b`, where the head of `secondaryPQ` corresponds to the best guess on the minimum distance between the element from the A index, `a`, and all of the elements in the B index.

The Hausdorff incremental join algorithm, shown in Figure 5.5, consists of a

```
Entry
   Element a;
   PriorityQueue secondaryPQ;
   float qMinUB;
```

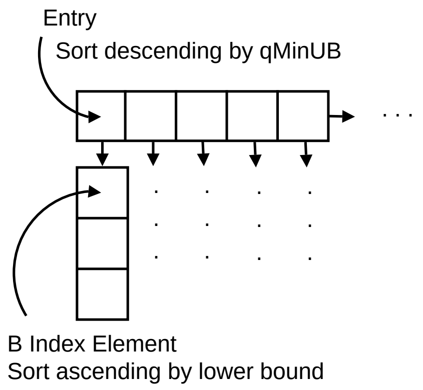


Figure 5.4: Each entry in the main priority queue corresponds to an element in the A index (a node or an object), and contains a reference to the element, `a`, a secondary priority queue of B index elements, `secondaryPQ`, and the minimum upper bound on the directed Hausdorff distance of the secondary priority queue, `qMinUB`.

while loop that continually pops entries from the main priority queue, `mainPQ`, and processes each entry by inserting child elements either into `mainPQ` or the entry's secondary queue. The algorithm is initialized on line 2 with one `Entry` that consists of the root of index A, `treeA`, and a secondary queue which consists of only the root of index B, `treeB`. The algorithm halts when the entry popped consists of a data object for `entry.a` and the top of the secondary priority queue, `b`, is also a data object, which is determined on line 6. Since the distance associated with the head of the secondary priority queue is the minimum distance between `a` and the set B, the head of the secondary priority queue will be the closest element in B to the data object `a`, that is, with the minimum distance between `a` and the B dataset. Furthermore, since the head of the main priority queue has the greatest `qMinUB` value (least upper bound), all other elements in the A dataset will have a lesser minimum distance to the B dataset, and therefore, the algorithm returns the

```
 1 HausdorffIncJoin(Index treeA, Index treeB) : float
 2   PriorityQueue mainPQ←initialize(treeA.root, treeB.root)
 3   while(mainPQ not empty) do
 4     Entry entry←mainPQ.pop();
 5     Element b←entry.secondaryPQ.top();
 6     if(entry.a is a leaf AND b is a leaf) then
 7       return dist(entry.a, b)
 8     endif;
 9
10     // expand node at the shallower level
11     // (leaves are at level 0)
12     if(entry.a.level >= b.level) then
13       // expand A
14       foreach child of entry.a do
15         Entry childEntry←new Entry(child);
16         createSecondary(childEntry, entry.secondaryPQ);
17         mainPQ.insert(childEntry);
18       enddo;
19     else
20       // expand B
21       expandSecondary(entry);
22       mainPQ.insert(entry)
23     endif
24   enddo;
25 end;
```

Figure 5.5: Function `HausdorffIncJoin` maintains the main priority queue. It halts when an entry containing two data objects is popped.

correct directed Hausdorff distance value.

On line 12 in Figure 5.5, the node to expand is chosen, which is either the node from the current entry of the main priority queue, `entry.a`, or the top of the secondary priority queue, `b`. In this case, since the goal is to reach two data objects, the node at the shallower level is chosen. If node `entry.a` from the A index is expanded, then, on line 15, an `Entry` is created for each child element of `entry.a`. On line 16, the `createSecondary` procedure (shown in Figure 5.6) is used to create the secondary priority queue for each new entry by calculating the bounds for the child on lines 3 and 4 of Figure 5.6. Elements are pruned by not adding them to the new secondary queue if the lower bound on the distance between `childEntry.a` and

```
createSecondary(childEntry, secondaryPQ)
  foreach b of secondaryPQ do
    float lowerBound←childEntry.a.minHausDist(b);
    float upperBound←childEntry.a.maxHausDist(b);
    if(lowerBound <= childEntry.qMinUB) then
      if( upperBound < childEntry.qMinUB ) then
        childEntry.qMinUB = upperBound;
      endif;
      childEntry.secondaryPQ.insert(b);
    endif;
  enddo;
end;
```

Figure 5.6: Given an entry from the main priority queue, procedure `createSecondary` calculates the bounds to each element in the given secondary priority query, `secondaryPQ`, and inserts them into the child entry's secondary priority queue if the lower bound is within the child entry's minimum upper bound, `qMinUB`.

`b` is greater than `childEntry.qMinUB`, which indicates that `b` could not possibly be or contain the data object from the B index with the minimal distance to any data object in `childEntry.a`.

If a node from the B index node is chosen for expansion, then the `expandSecondary` procedure is called on line 21 of Figure 5.5 to expand the head of the secondary queue of `entry`, which is then reinserted into `mainPQ` on line 22. The `expandSecondary` procedure, shown in Figure 5.7, pops the first element of the secondary queue, `b`, and expands it by adding each of its children, `c`, to the secondary queue. Again, elements are pruned by not adding them to the queue if the lower bound distance between `entry.a` and `c` is greater than `qMinUB`, as was done with the `createSecondary` procedure.

To calculate the partial Hausdorff distance, the `HausdorffIncJoin` function is modified to output the distance between the object from set A with the k^{th} minimal

```
expandSecondary(Entry entry)
  Element b←entry.secondaryPQ.remove();
  foreach child, c of b do
    float lowerBound←entry.a.minHausDist(c);
    if(lowerBound<=entry.qMinUB) then
      entry.secondaryPQ.insert(c);
    endif;
  enddo;
end;
```

Figure 5.7: The `expandSecondary` procedure expands node `b`, the top of the secondary priority queue, `entry.secondaryPQ`. Each child of `b` is inserted into `entry.secondaryPQ` if its lower bound is less than or equal to the entry's minimum upper bound, `qMinUB`.

Hausdorff distance. Instead of returning the distance between the first pair of objects on line 7 of Figure 5.5, the distance between the k^{th} pair is returned. The secondary queue facilitates this semi-join operation by removing the `Entry` at the head of the main queue, which removes all references to object `a`.

The algorithm also needs to be modified if the sets are so large that the queues overfill internal memory. See [83] for a discussion of managing priority queues under such conditions.

5.4 Experiments

To test the effectiveness of the incremental distance join approach, two different geographic datasets were used: lake shape files from the Ontario MNDM [142] and polygon data from the Sequoia benchmark dataset [182]. The code was developed in Java and the experiments were run on a Compaq computer with an AMD Athlon 64 Processor and 896 megabytes of RAM.

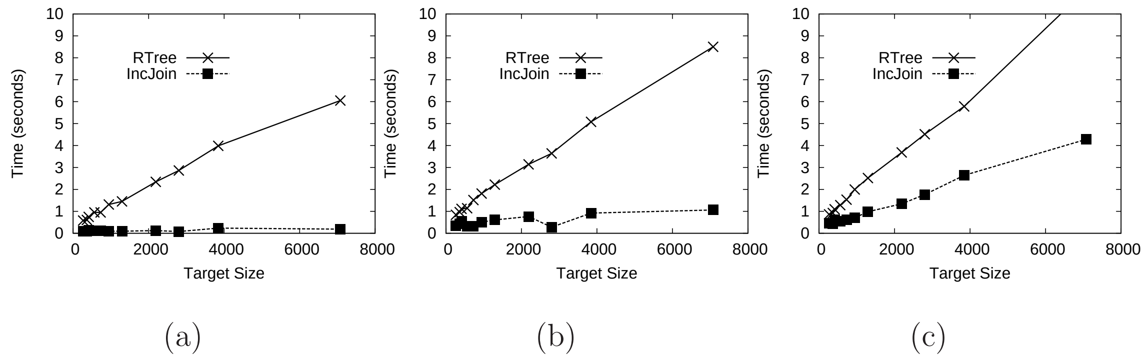


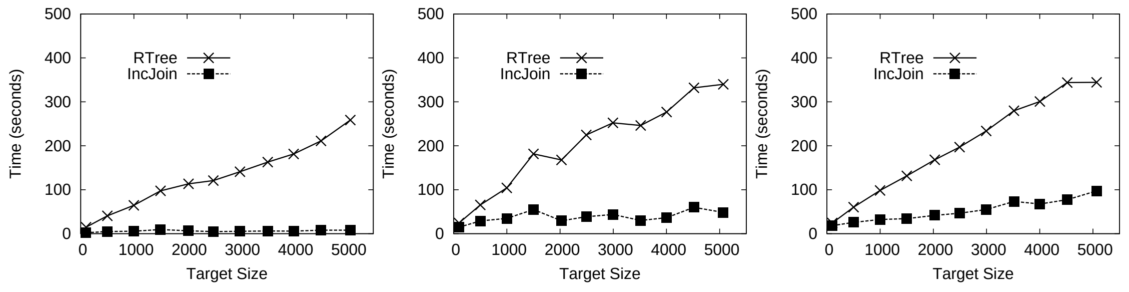
Figure 5.8: The time to calculate the Hausdorff distance against every other image for a target image using (a) the Ontario lakes data, (b) the translated lakes data, and (c) the normalized lakes data.

For the traditional approach, instead of using a Voronoi diagram to index the data, an R-tree that indexes the vertices of the polygons was used, which was then probed, that is, the B dataset was indexed, and then for every object in the A dataset, its nearest neighbor in the B set was found. The maximum of these values over the A dataset is the directed Hausdorff distance.

The experiments were run as image matching exercises. One object from the dataset was chosen as the target image, and then the Hausdorff distance to every other object in the dataset was calculated by finding the directed Hausdorff distance between the target and the test image, and then finding the directed Hausdorff distance between the test and the target, with the Hausdorff distance being the maximum of the two values. Each object was indexed using a basic R-tree [76] in internal memory with a low fan-out for performance. This build time is not included in the performance plots since both approaches build an R-tree on each object.

The results of the experiments with the Ontario MNDM lakes dataset containing 145 objects are shown in Figure 5.8. Target images of varying sizes (number of vertices) were chosen. The first plot, Figure 5.8a, shows the time to calculate the Hausdorff distance to every other object, while the second and third plots, Figures 5.8b and c, show the time needed when the data was translated and normalized, respectively. With the translated data, each data object is centered at the point with (0,0) coordinate values, without resizing the objects. With the normalized data, each object is translated and resized to have a width and height of one. The translation and normalization of the data was performed off line. As shown in Figure 5.8a, with the raw data, the incremental distance join quickly calculates the Hausdorff distance for any size image with near constant time, while the traditional approach degrades with larger target sizes. In this case, since the objects are disjoint, there will only be a few vertices on the far sides of the objects that could potentially have the maximum minimum distance, and the incremental distance join only needs to distinguish these few vertices, while the traditional approach must still find the minimum for every vertex. With the translated data, every vertex could potentially have the maximum minimum distance, and, as shown in Figure 5.8b, it takes slightly longer for both approaches to calculate the Hausdorff distance. Finally, with the normalized data in Figure 5.8c, both approaches degrade and the incremental distance no longer performs with constant time, though still better than the traditional approach.

Figure 5.9 shows the results of experiments performed using a portion of the Sequoia dataset [182]. The observed performance was similar. Note that in these experiments, only the larger polygons were used. In particular, only polygons with



(a) (b) (c)

Figure 5.9: The time to calculate the Hausdorff distance against every other image for a target image using (a) the larger polygons from the Sequoia dataset, (b) the translated data, and (c) the normalized data.

over 100 vertices, thereby resulting in 5,294 objects in the dataset. Again, the incremental distance join performed in nearly constant time for the raw and translated data. Only with the normalized data, Figure 5.9c, did the performance degrade with size, but only slightly.

Chapter 6

Conclusions and Future Work

In Chapter 2, we provided an in-depth survey and analysis of the various techniques used to perform a spatial join using a filter-and-refine approach in which complex objects are approximated, typically by a minimum bounding rectangle. The approximations are joined, producing a candidate set which is refined to produce the final results using the full objects. We examined various techniques for performing a spatial join using the available internal memory, using either nested-loop joins, indexed nested-loop joins, or variants of the plane-sweep technique. If there is insufficient internal memory, then external memory can be used to process larger datasets. We examined cases where the data is indexed or not indexed. If the datasets are indexed, then overlapping data pages can be read in a predetermined order or the two indices can be traversed synchronously if the indices are hierarchical. Pairs of overlapping data pages are joined using internal memory techniques. Alternatively, if the data is not indexed, then the data can be partitioned using a variety of techniques, and then overlapping partition pairs can be joined using internal memory techniques. We also examined the techniques and issues involved with refining the candidate set produced during the filtering stage. Finally, we looked at spatial joins in a variety of situations, such as multiway spatial joins, parallel spatial joins, and distributed spatial joins.

At this point, we mention that our main goal with the survey was to provide a guide as to what techniques work best in particular situations. However, we were not able to conclusively determine which techniques are superior for each scenario as we observed that the experiments comparing techniques were inconclusive and could easily be skewed by the choice of experimental data or details of implementation. Nevertheless, we feel that this survey illuminates some of these issues and hope that it motivates further analyses and experimentation.

In addition, the analysis in the survey recasts the design of the spatial join algorithms in a new light, which gives rise to many interesting questions to be answered in future work, such as:

1. What is the best duplicate removal method?
2. What is the best repartitioning method?
3. When are methods for unindexed data better than those that first build an index?

In Chapter 3, we introduced a novel algorithm called the Iterative Spatial Join for performing a spatial join that operates similarly to the SSSJ algorithm [19], but without its drawbacks, and for some datasets, significantly better than the PBSM [155] and S3J [116] algorithms. SSSJ, PBSM, and S3J are three of the best known algorithms for the task of performing spatial joins on large unindexed datasets requiring the use of external memory. Furthermore, the Iterative Spatial Join provides a natural extension to the traditional plane-sweep algorithm, which is the preferred method for performing spatial joins for smaller datasets. In particular, the algorithm makes additional passes on the data when there is not enough internal memory to store the data structures of the plane-sweep algorithm. This is in contrast to the SSSJ algorithm which must invoke a partitioning method in such a case, thereby ignoring all the work done before. Thus the Iterative Spatial Join algorithm can be said to be a true "external memory" spatial join method in that it accounts for both the case that internal memory is not large enough to store the data, and the case that internal memory is not large enough to store the internal data structures to make the method work. The fact that we do not need to guess when the internal memory is large enough for the sweep structures makes the performance of the Iterative Spatial Join method more predictable than the SSSJ method.

We have also shown the effect of density on spatial join algorithms. The density is the number of objects intersecting a sweep line divided by the number of objects. The average density is the average over all stops of the sweep line and the maximum density is the maximum over all of the stops. We have demonstrated that the performance of an algorithm is inversely related to the density of a dataset. In particular, datasets with many more objects effect the performance of the Iterative Spatial Join algorithm in a predictable pattern, while this is not necessarily the case for the other algorithms.

In our experiments, we implemented the best known versions of the SSSJ, S3J, and PBSM algorithms and we subjected them to a rigorous set of tests. However, we by no means feel that we have definitively determined the superior algorithm. While we have compared the best known algorithms, there may yet be better algorithms developed or more promising variations. While building indices on the fly has been shown to be an inferior approach [155], there are many types of spatial indices and one might prove fast enough to build. Even with the algorithms that we have compared, there are still many factors to explore to definitively answer which algorithm is the best. While we have used what we feel to be the best plane-sweep implementation, there are other variations [19, 50], but there are very few experiments comparing the performance of the algorithms using different plane-sweep methods. Furthermore, while we have tried to give each algorithm every advantage, the effects of tuning parameters and machine parameters (I/O speed, CPU speed, etc.) impact performance and further study of how these parameters effect the algorithms is required.

One approach that could be used to improve spatial join processing would be to combine the PBSM and Iterative Spatial Join algorithms. If a PBSM style partitioning were to be performed prior to using the Iterative Spatial Join algorithm, then the best performance would be seen for low density datasets. If there was insufficient memory, then the Iterative Spatial Join algorithm would naturally use its external memory algorithm to finish processing the data.

As for the testing methods, we feel that the standard test datasets (Sequoia [182] and the U.S. Bureau of the Census [190] Tiger data), while adequate for demonstrating the usefulness of an algorithm, fail to examine the full operating characteristics of an algorithm. We therefore used generated datasets as a means to better explore the algorithms so that they might be applied in a variety of situations, especially those that we have not anticipated.

In Chapter 4, we described the Quickjoin algorithm, a novel method for performing a similarity join, and shown it to be highly competitive. The algorithm works by partitioning the data using only the metric-space distance information, rather than using any vector-format information. In effect, the algorithm creates partitions using the data rather than partitioning based on space, as done in many of the multi-dimensional indices used in existing similarity join algorithms.

The method overcomes deficiencies that occur with many existing methods that rely on multi-dimensional indices. For larger values of ϵ, multi-dimensional indices are often not useful for higher dimensional similarity joins since a large value of ϵ will cause the index to return all or nearly all of the values in the index, thereby resulting in poor performance. In addition, if an index is not used, then there is more flexibility in specifying distance functions, thereby allowing the distance (similarity) function to be changed dynamically.

In Chapter 5, we presented a method that improves the speed of calculating the Hausdorff distance over traditional approaches. The technique uses a modified version of an incremental distance join [83, 179, 180, 206] that requires more complicated calculations to find the lower and upper bounds on the Hausdorff distance between index nodes. Experiments verified the improved performance.

Future work involves investigating the performance of the algorithm when the objects are rotated to find better matches, finding the k^{th} partial match in order to exclude outliers, as well as other application areas.

Other future work involves exploring particular applications in more detail, specifically, computational genomics. Both the Quickjoin and the Hausdorff incremental distance join have applications in data mining, and exploring the applications of these algorithms, as well as related algorithms, to a specific domain should provide topics for interesting further research.

Bibliography

[1] D. J. Abel, V. Gaede, R. Power, and X. Zhou. Caching strategies for spatial joins. *GeoInformatica*, 3(1):33–59, June 1999.

[2] D. J. Abel, B. C. Ooi, K.-L. Tan, R. Power, and J. X. Yu. Spatial join strategies in distributed spatial DBMS. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases—4th International Symposium, SSD'95*, vol. 951 of Springer-Verlag Lecture Notes in Computer Science, pages 348–367, Portland, ME, August 1995.

[3] D. J. Abel and J. L. Smith. A data structure and query algorithm for a database of areal entities. *Australian Computer Journal*, 16(4):147–154, November 1984.

[4] C. C. Aggarwal. Towards systematic design of distance functions for data mining applications. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9–18, Washington, D.C., August 2003.

[5] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In D. B. Lomet, editor, *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, vol. 730 of Springer-Verlag Lecture Notes in Computer Science, pages 69–84, Chicago, October 1993.

[6] R. Agrawal, G. Psaila, E. L. Wimmers, and M. Zaït. Querying shapes of histories. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings of 21st VLDB International Conference on Very Large Data Bases*, pages 502–514, Zurich, Switzerland, September 1995.

[7] H. Alt, O. Aichholzer, and Günter Rote. Matching shapes with a reference point. In *SCG '94: Proceedings of the Tenth Annual Symposium on Computational Geometry*, pages 85–92, Stony Brook, NY, June 1994.

[8] H. Alt, B. Behrends, and J. Blömer. Approximate matching of polygonal shapes (extended abstract). In *SCG '91: Proceedings of the Seventh Annual Symposium on Computational Geometry*, pages 186–193, North Conway, NH, June 1991.

[9] N. An, Z.-Y. Yang, and A. Sivasubramaniam. Selectivity estimation for spatial joins. In *Proceedings of the 17th IEEE International Conference on Data Engineering*, pages 368–375, Heidelberg, Germany, April 2001.

[10] Carlos Andújar, Pere Brunet, and Dolors Ayala. Topology-reducing surface simplification using a discrete solid representation. *ACM Transactions on Graphics*, 21(2):88–105, April 2002.

[11] W. G. Aref and I. F. Ilyas. An extensible index for spatial databases. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, pages 49–58, Fairfax, VA, July 2001.

[12] W. G. Aref and H. Samet. Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 178–189, Charleston, SC, August 1992.

[13] W. G. Aref and H. Samet. A cost model for query optimization using R-trees. In N. Pissinou and K. Makki, editors, *Proceedings of the 2nd ACM Workshop on Geographic Information Systems*, pages 60–67, Gaithersburg, MD, December 1994.

[14] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM)*, pages 347–354, Gaithersburg, MD, December 1994.

[15] W. G. Aref and H. Samet. The spatial filter revisited. In T. C. Waugh and R. G. Healey, editors, *Proceedings of the 6th International Symposium on Spatial Data Handling*, pages 190–208, Edinburgh, Scotland, September 1994. International Geographical Union Commission on Geographic Information Systems, Association for Geographical Information.

[16] W. G. Aref and H. Samet. Cascaded spatial join algorithms with spatially sorted output. In S. Shekhar and P. Bergougnoux, editors, *Proceedings of the 4th ACM Workshop on Geographic Information Systems*, pages 17–24, Gaithersburg, MD, November 1996.

[17] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In M. T. Goodrich and C. C. McGeoch, editors, *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation (ALENEX'99)*, vol. 1619 of Springer-Verlag Lecture Notes in Computer Science, pages 328–348, Baltimore, MD, January 1999.

[18] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. In C. Zaniolo, P. C. Lockemann, M. H. Scholl, and T. Grust, editors, *Proceedings of the 7th International Conference on Extending Database Technology—EDBT 2000*, vol. 1777 of Springer-Verlag Lecture Notes in Computer Science, pages 413–429, Konstanz, Germany, March 2000.

[19] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In A. Gupta, O. Shmueli, and J. Widom, editors, *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 570–581, New York, August 1998.

[20] W.M. Badawy and W.G. Aref. On local heuristics to speed up polygon-polygon intersection tests. In Claudia Bauzer Medeiros, editor, *Proceedings of the 7th ACM International Symposium on Advances in Geographic Information Systems*, pages 97–102, Kansas City, MO, November 1999.

[21] I. J. Balaban. An optimal algorithm for finding segments intersections. In *SCG '95: Proceedings of the Eleventh Annual Symposium on Computational Geometry*, pages 211–219, Vancouver, British Columbia, Canada, June 1995.

[22] D. H. Ballard. Strip trees: a hierarchical representation for curves. *Communications of the ACM*, 24(5):310–321, May 1981. Also see corrigendum, *Communications of the ACM*, 25(3):213, March 1982.

[23] L. Becker, A. Giesen, K. Hinrichs, and J. Vahrenhold. Algorithms for performing polygonal map overlay and spatial join on massive data set. In R. H. Güting, D. Papadias, and F. H. Lochovsky, editors, *Advances in Spatial Databases—6th International Symposium, SSD'99*, vol. 1651 of Springer-Verlag Lecture Notes in Computer Science, pages 270–285, Hong Kong, China, July 1999.

[24] L. Becker, K. Hinrichs, and U. Finke. A new algorithm for computing joins with grid files. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, pages 190–197, Vienna, Austria, April 1993.

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

[26] A. Belussi, E. Bertino, and A. Nucita. Grid based methods for estimating spatial join selectivity. In I. F. Cruz and D. Pfoser, editors, *Proceedings of the 12th ACM International Workshop on Advances in Geographic Information Systems*, pages 92–100, Washington, DC, November 2004.

[27] A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 299–310, Zurich, Switzerland, September 1995.

[28] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[29] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.

[30] C. Böhm, B. Braunmüller, M. Breunig, and H.-P. Kriegel. High performance clustering based on the similarity join. In *Proceedings of the 9th CIKM International Conference on Information and Knowledge Management*, pages 298–305, McLean, VA, November 2000.

[31] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. In *Proceedings of the ACM SIGMOD Conference*, pages 379–390, Santa Barbara, CA, May 2001.

[32] C. Böhm and H.-P. Kriegel. A cost model and index architecture for the similarity join. In *Proceedings of the 17th IEEE International Conference on Data Engineering*, pages 411–420, Heidelberg, Germany, April 2001.

[33] S. Brin. Near neighbor search in large metric spaces. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 574–584, Zurich, Switzerland, September 1995.

[34] T. Brinkhoff and H.-P. Kriegel. Approximations for a multi-step processing of spatial joins. In J. Nievergelt, T. Roos, H.-J. Schek, and P. Widmayer, editors, *IGIS'94: Geographic Information Systems, International Workshop on Advanced Research in Geographic Information Systems*, pages 25–34, Monte Verità, Ascona, Switzerland, March 1994.

[35] T. Brinkhoff and H.-P. Kriegel. The impact of global clustering on spatial database systems. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 168–179, Santiago, Chile, September 1994.

[36] T. Brinkhoff, H.-P. Kriegel, and R. Schneider. Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, pages 40–49, Vienna, Austria, April 1993.

[37] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proceedings of the ACM SIGMOD Conference*, pages 197–208, Minneapolis, MN, June 1994.

[38] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the ACM SIGMOD Conference*, pages 237–246, Washington, DC, May 1993.

[39] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using R-trees. In S. Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering*, pages 258–265, New Orleans, LA, February 1996.

[40] A. Brinkmann and K. Hinrichs. Implementing exact line segment intersection in map overlay. In T. K. Poiker and N. Chrisman, editors, *Proceedings of the 8th International Symposium on Spatial Data Handling*, pages 569–579, GIS Lab, Department of Geography, Simon Fraser University, Burnaby, British

Columbia, Canada, July 1998. International Geographical Union, Geographic Information Science Study Group.

[41] A. Brodsky, C. Lassez, J. Lassez, and M. J. Maher. Separability of polyhedra for optimal filtering of spatial and constraint data. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 54–65, San Jose, CA, May 1995.

[42] V. Chalana and Y. Kim. A methodology for evaluation of boundary detection algorithms on medical images. *IEEE Transactions on Medical Imaging*, 16(5):642–652, October 1997.

[43] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–322, September 2001. Also University of Chile DCC Technical Report TR/DCC-99-3, June 1999.

[44] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, January 1992.

[45] N. Chiba and T. Nishizeki. The hamiltonian cycle problem is linear-time solvable for 4-connect planar graphs. *Journal of Algorithms*, 10(2):187–211, June 1989.

[46] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, pages 173–174. MIT Press/McGraw-Hill, Cambridge, MA, 1990.

[47] A. Corral, M. Vassilakopoulos, and Y. Manolopoulos. Algorithms for joining R-trees and linear region quadtrees. In R. H. Güting, D. Papadias, and F. H. Lochovsky, editors, *Advances in Spatial Databases—6th International Symposium, SSD'99*, vol. 1651 of Springer-Verlag Lecture Notes in Computer Science, pages 251–269, Hong Kong, China, July 1999.

[48] A. Das, J. Gehrke, and M. Riedewald. Approximation techniques for spatial data. In *Proceedings of the ACM SIGMOD Conference*, pages 695–706, Paris, France, June 2004.

[49] M. B. Dillencourt and H. Samet. Using topological sweep to extract the boundaries of regions in maps represented by region quadtrees. *Algorithmica*, 15(1):82–102, January 1996. Also see *Proceedings of the Third International Symposium on Spatial Data Handling*, pages 65–77, Sydney, Australia, August 1988 and University of California at Irvine Information and Computer Science Technical Report ICS TR 91-01.

[50] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 535–546, San Diego, CA, February 2000.

[51] J.-P. Dittrich and B. Seeger. GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 47–56, San Francisco, California, August 2001.

[52] D. Dori and M. Ben-Bassat. Circumscribing a convex polygon by a polygon of fewer sides with minimal area addition. *Computer Vision, Graphics, and Image Processing*, 24(2):131–159, November 1983.

[53] C. A. Duncan, M. Goodrich, and S. Kobourov. Balanced aspect ratio trees: combining the advantages of k-d trees and octrees. *Journal of Algorithms*, 38(1):303–333, January 2001. Also see *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 300–309, Baltimore, MD, January 1999.

[54] H. Edelsbrunner. Dynamic rectangle intersection searching. Institute for Information Processing Technical Report 47, Technical University of Graz, Graz, Austria, February 1980.

[55] H. Edelsbrunner. A new approach to rectangle intersections: part I. *International Journal of Computer Mathematics*, 13(3–4):209–219, 1983.

[56] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, Reading, MA, third edition, 2000.

[57] J. Enderle, M. Hampel, and T. Seidl. Joining interval data in relational databases. In *Proceedings of the ACM SIGMOD Conference*, pages 683–694, Paris, France, June 2004.

[58] C. Esperança and H. Samet. Spatial database programming using SAND. In M. J. Kraak and M. Molenaar, editors, *Proceedings of the 7th International Symposium on Spatial Data Handling*, volume 2, pages A29–A42, Delft, The Netherlands, August 1996. International Geographical Union Commission on Geographic Information Systems, Association for Geographical Information.

[59] C. Esperança and H. Samet. Orthogonal polygons as bounding structures in filter-refine query processing strategies. In M. Scholl and A. Voisard, editors, *Advances in Spatial Databases—5th International Symposium, SSD'97*, vol. 1262 of Springer-Verlag Lecture Notes in Computer Science, pages 197–220, Berlin, Germany, July 1997.

[60] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, 1994.

[61] C. Faloutsos and I. Kamel. Beyond uniformity and independence: analysis of R-trees using the concept of fractal dimension. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 4–13, Minneapolis, MN, May 1994.

[62] C. Faloutsos and K.-I. Lin. FastMap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the ACM SIGMOD Conference*, pages 163–174, San Jose, CA, May 1995.

[63] C. Faloutsos, B. Seeger, A. J. M. Traina, and C. Traina Jr. Spatial join selectivity using power laws. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *Proceedings of the ACM SIGMOD Conference*, pages 177–188, Dallas, TX, May 2000.

[64] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[65] J. D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, MA, 1982.

[66] F. Fotouhi and S. Pramanik. Optimal secondary storage access sequence for performing relational join. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):318–328, September 1989.

[67] V. Gaede. Optimal redundancy in spatial database systems. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases—4th International Symposium, SSD'95*, vol. 951 of Springer-Verlag Lecture Notes in Computer Science, pages 96–116, Portland, ME, August 1995.

[68] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 20(2):170–231, June 1998. Also International Computer Science Institute Report TR–96–043, October 1996.

[69] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, Englewood-Cliffs, NJ, 2000.

[70] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: a hierarchical structure for rapid interference detection. In *Proceedings of the SIGGRAPH'96 Conference*, pages 171–180, New Orleans, LA, August 1996.

[71] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[72] O. Günther. Efficient computation of spatial joins. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, pages 50–59, Vienna, Austria, April 1993.

[73] O. Günther, V. Oria, P. Picouet, J.-M. Saglio, and M. Scholl. Benchmarking spatial joins à la carte. In M. Rafanelli and M. Jarke, editors, *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 32–41, Capri, Italy, July 1998.

[74] C. Gurret and P. Rigaux. The sort/sweep algorithm: A new method for R-tree based spatial joins. In *Proceedings of the 12th International Conference on Statistical and Scientific Database Management (SSDBM)*, pages 153–165, Berlin, Germany, July 2000.

[75] R. H. Güting and W. Schilling. A practical divide-and-conquer algorithm for the rectangle intersection problem. *Information Sciences*, 42(2):95–112, July 1987.

[76] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, June 1984.

[77] E. N. Hanson. The interval skip list: a data structure for finding all intervals that overlap a point. Computer Science and Engineering Technical Report WSU–CS–91–01, Wright State University, Dayton, OH, 1991.

[78] L. Harada, M. Nakano, M. Kitsuregawa, and M. Takagi. Query processing for multi-attribute clustered records. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases*, pages 59–70, Brisbane, Queensland, Australia, August 1990.

[79] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 562–573, Zurich, Switzerland, September 1995.

[80] A. Henrich and J. Möller. Extending a spatial access structure to support additional standard attributes. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases—4th International Symposium, SSD'95*, vol. 951 of Springer-Verlag Lecture Notes in Computer Science, pages 132–151, Portland, ME, August 1995.

[81] A. Henrich, H.-W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional point and non-point data. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, pages 45–53, Amsterdam, The Netherlands, August 1989.

[82] S. Hettich and S. D. Bay. The UCI KDD archive [`http://kdd.ics.uci.edu`]. Irvine, CA: University of California, Department of Information and Computer Science, 1999.

[83] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In L. Hass and A. Tiwary, editors, *Proceedings of the ACM SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[84] G. R. Hjaltason and H. Samet. Improved bulk-loading algorithms for quadtrees. In Claudia Bauzer Medeiros, editor, *Proceedings of the 7th ACM*

International Symposium on Advances in Geographic Information Systems, pages 110–115, Kansas City, MO, November 1999.

[85] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, December 2003.

[86] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, February 1962.

[87] E. Hoel and H. Samet. Data-parallel spatial join algorithms. In *Proceedings of the 23rd International Conference on Parallel Processing*, volume 3, pages 227–234, St. Charles, IL, August 1994.

[88] E. G. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 606–618, Zurich, Switzerland, September 1995.

[89] G. Hristescu and M. Farach-Colton. Cluster-preserving embedding of proteins. Technical report, Department of Computer Science, Rutgers University, Piscataway, NJ, 1999.

[90] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. A cost model for estimating the performance of spatial joins using R-trees. In Y. E. Ioannidis and D. M. Hansen, editors, *Proceedings of the 9th International Conference on Scientific and Statistical Database Management*, pages 30–38, Olympia, WA, August 1997.

[91] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using R-trees: breadth-first traversal with global optimizations. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 396–405, Athens, Greece, August 1997.

[92] Y.-W. Huang, M. Jones, and E. A. Rundensteiner. Improving spatial intersect joins using symbolic intersect detection. In M. Scholl and A. Voisard, editors, *Advances in Spatial Databases—5th International Symposium, SSD'97*, vol. 1262 of Springer-Verlag Lecture Notes in Computer Science, pages 165–177, Berlin, Germany, July 1997.

[93] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[94] D. P. Huttenlocher, K. Kedem, and J. M. Kleinberg. On dynamic voronoi diagrams and the minimum Hausdorff distance for point sets under euclidean motion in the plane. In *SCG '92: Proceedings of the Eighth Annual Symposium on Computational Geometry*, pages 110–119, Berlin, Germany, June 1992.

[95] D. P. Huttenlocher, D. A. Klanderman, and W. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, September 1993.

[96] P. Indyk and S. Venkatasubramanian. Approximate congruence in nearly linear time. In *SODA '00: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 354–360, San Francisco, CA, January 2000.

[97] G. S. Iwerks and H. Samet. The spatial spreadsheet. In D. P. Huijsmans and A. W. M. Smeulders, editors, *Proceedings of the 3rd International Conference on Visual Information Systems (VISUAL99)*, pages 317–324, Amsterdam, The Netherlands, June 1999.

[98] E. Jacox and H. Samet. Iterative spatial join. *ACM Transactions on Database Systems*, 28(3):268–294, September 2003.

[99] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the ACM SIGMOD Conference*, pages 332–342, Atlantic City, NJ, June 1990.

[100] H. V. Jagadish. Spatial search with polyhedra. In *Proceedings of the 6th IEEE International Conference on Data Engineering*, pages 311–319, Los Angeles, February 1990.

[101] J. Ja'Ja'. A perspective on quicksort. *Computing in Science and Engineering*, 2(1):43–49, January 2000.

[102] T. Kahveci, C. Lang, and A. K Singh. Joining massive high-dimensional datasets. In *Proceedings of the 19th IEEE International Conference on Data Engineering*, pages 264–276, Bangalore, India, March 2003.

[103] D. Kalashnikov and S. Prabhakar. Similarity joins for low- and high- dimensional data. In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA'03)*, pages 7–16, Kyoto, Japan, March 2003.

[104] I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM)*, pages 490–499, Washington, DC, November 1993.

[105] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In J. Peckham, editor, *Proceedings of the ACM SIGMOD Conference*, pages 369–380, Tucson, AZ, May 1997.

[106] G. Kedem. The quad-cif tree: a data structure for hierarchical on-line algorithms. Computer Science Technical Report TR–91, University of Rochester, Rochester, NY, September 1981.

[107] G. Kedem. The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference*, pages 352–357, Las Vegas, NV, June 1982. Also University of Rochester Computer Science Technical Report TR–91, September 1981.

[108] K. Kedem and Y. Yarmovski. Curve based stereo matching using the minimum Hausdorff distance. In *SCG '96: Proceedings of the Twelfth Annual Symposium on Computational Geometry*, pages 415–418, Philadelphia, PA, May 1996.

[109] S.-W. Kim, W.-S. Cho, M.-J. Lee, and K.-Y. Whang. A new algorithm for processing joins using the multilevel grid file. In T. W. Ling and Y. Masunaga, editors, *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA'95)*, volume 5, pages 115–123, Singapore, April 1995.

[110] M. Kitsuregawa, L. Harada, and M. Takagi. Join strategies on KD-tree indexed relations. In *Proceedings of the 5th IEEE International Conference on Data Engineering*, pages 85–93, Los Angeles, February 1989.

[111] A. Klinger. Patterns and search statistics. In J. S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 303–337. Academic Press, New York, 1971.

[112] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998.

[113] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, 1973.

[114] K. Koperski and J. Han. Discovery of spatial association rules in geographic information systems. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases—4th International Symposium, SSD'95*, vol. 951 of Springer-Verlag Lecture Notes in Computer Science, pages 47–66, Portland, ME, August 1995.

[115] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, pages 215–226, Mumbai (Bombay), India, September 1996.

[116] N. Koudas and K. C. Sevcik. Size separation spatial join. In J. Peckham, editor, *Proceedings of the ACM SIGMOD Conference*, pages 324–335, Tucson, AZ, May 1997.

[117] N. Koudas and K. C. Sevcik. High dimensional similarity joins: algorithms and performance evaluation. In *Proceedings of the 14th IEEE International Conference on Data Engineering*, pages 466–475, Orlando, FL, February 1998.

[118] N. Koudas and K. C. Sevcik. High dimensional similarity joins: algorithms and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 12(1):3–18, January/February 2000.

[119] H.-P. Kriegel, P. Kunath, M. Pfeifle, and M. Renz. Spatial join for high-resolution objects. In *Proceedings of the 16th (IEEE) International Conference on Scientific and Statistical Database Management (SSDBM'04)*, pages 151–160, Santorini Island, Greece, June 2004.

[120] V. Kumar. Algorithms for constraints satisfaction problems: A survey. *The AI Magazine, by the AAAI*, 13(1):32–44, Spring 1992.

[121] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15:215–245, 1995. Also see *Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science*, pages 577–591, Santa Fe, NM, November 1994.

[122] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the ACM SIGMOD Conference*, pages 209–220, Minneapolis, MN, June 1994.

[123] M.-L. Lo and C. V. Ravishankar. Generating seeded trees from data sets. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases—4th International Symposium, SSD'95*, vol. 951 of Springer-Verlag Lecture Notes in Computer Science, pages 328–347, Portland, ME, August 1995.

[124] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proceedings of the ACM SIGMOD Conference*, pages 247–258, Montréal, Canada, June 1996.

[125] H. Lu, R. Luo, and B. C. Ooi. Spatial joins by precomputation of approximations. In *Proceedings of the 6th Australasian Database Conference*, Australian Computer Science Communications, volume 17, number 2, pages 132–142, Glenelg, South Australia, Australia, January 1995. Also in *Australian Computer Science Communications*, 17(2):143–152, January 1995.

[126] G. Luo, J. F. Naughton, and C. Ellmann. A non-blocking parallel spatial join algorithm. In *Proceedings of the 18th International Conference on Data Engineering*, pages 697–705, San Jose, CA, February 2002.

[127] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 307–325. Springer-Verlag, Berlin, West Germany, 1988.

[128] N. Mamoulis, P. Kalnis, S. Bakiras, and X. Li. Optimization of spatial joins on mobile devices. In *Advances in Spatial and Temporal Databases : 8th International Symposium, SSTD*, pages 233–251, Santorini Island, Greece, July 2003.

[129] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *Proceedings of the ACM SIGMOD Conference*, pages 1–12, Philadelphia, PA, June 1999.

[130] N. Mamoulis and D. Papadias. Multiway spatial joins. *ACM Transactions on Database Systems*, 26(4):424–475, December 2001.

[131] N. Mamoulis and D. Papadias. Selectivity estimation of complex spatial queries. In *Advances in Spatial and Temporal Databases : 7th International Symposium, SSTD*, pages 155–174, Redondo Beach, CA, July 2001.

[132] N. Mamoulis and D. Papadias. Slot index spatial join. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):211–231, 2003.

[133] N. Mamoullis and D. Papadias. Constraint-based algorithms for computing clique intersection joins. In R. Laurini, K. Makki, and N. Pissinou, editors, *Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems*, pages 118–123, Washington, DC, November 1998.

[134] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985. Also Xerox Palo Alto Research Center Technical Report CSL–81–5, January 1982.

[135] T. H. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling of page-fetches in join operations. In *Very Large Data Bases, 7th International Conference*, pages 488–498, Cannes, France, September 1981.

[136] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[137] D. M. Mount, N. S. Netanyahu, and J. Le Moigne. Improved algorithms for robust point pattern matching and applications to image registration. In *SCG '98: Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 155–164, June 1998.

[138] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proceedings of the ACM SIGMOD Conference*, pages 270–277, San Francisco, May 1987.

[139] G. Neyer and P. Widmayer. Singularities make spatial join scheduling hard. In *Algorithms and Computation, 8th International Symposium, ISAAC*, pages 293–302, Singapore, December 1997.

[140] 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.

[141] S. M. Omohundro. Five balltree construction algorithms. Technical Report TR–89–063, International Computer Science Institute, Berkeley, CA, December 1989.

[142] Ontario Ministry of Northern Development and Mines, Mining Lands Section. Ontario mining land tenure spatial data. Technical report, 2006.

[143] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the ACM SIGMOD Conference*, pages 326–336, Washington, DC, May 1986.

[144] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 294–305, Portland, OR, June 1989.

[145] J. A. Orenstein. Strategies for optimizing the use of redundancy in spatial databases. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, editors, *Design and Implementation of Large Spatial Databases—1st Symposium, SSD'89*, vol. 409 of Springer-Verlag Lecture Notes in Computer Science, pages 115–134, Santa Barbara, CA, July 1989.

[146] J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14(5):611–629, May 1988.

[147] T. Ottmann and D. Wood. Space-economical plane-sweep algorithms. *Computer Vision, Graphics, and Image Processing*, 34(1):35–51, April 1986.

[148] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.

[149] D. Papadias and D. Arkoumanis. Approximate processing of multiway spatial joins in very large databases. In C. S. Jensen, K. G. Jeffery, J. Pokorný, S. Saltenis, E. Bertino, K. Böhm, and M. Jarke, editors, *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology*, volume 2287 of *Lecture Notes in Computer Science*, pages 179–196, Prague, Czech Republic, March 2002.

[150] D. Papadias, N. Mamoulis, and V. Delis. Algorithms for querying by spatial structure. In A. Gupta, O. Shmueli, and J. Widom, editors, *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 546–557, New York, August 1998.

[151] D. Papadias, N. Mamoulis, and Y. Theodoridis. Processing and optimization of multiway spatial joins using R-trees. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 44–55, Philadelphia, PA, May–June 1999.

[152] A. Papadopoulos, P. Rigaux, and M. Scholl. A performance evaluation of spatial join processing strategies. In R. H. Güting, D. Papadias, and F. H. Lochovsky, editors, *Advances in Spatial Databases—6th International Symposium, SSD'99*, vol. 1651 of Springer-Verlag Lecture Notes in Computer Science, pages 286–307, Hong Kong, China, July 1999.

[153] H.-H. Park, G.-H. Cha, and C.-W. Chung. Multi-way spatial joins using R-trees: methodology and performance evaluation. In R. H. Güting, D. Papadias, and F. H. Lochovsky, editors, *Advances in Spatial Databases—6th International Symposium, SSD'99*, vol. 1651 of Springer-Verlag Lecture Notes in Computer Science, pages 229–250, Hong Kong, China, July 1999.

[154] H.-H. Park, C.-G. Lee, Y.-J. Lee, and C.-W. Chung. Early separation of filter and refinement steps in spatial query optimization. In A. L. P. Chen and F. H. Lochovsky, editors, *Database Systems for Advanced Applications, Proceedings of the Sixth International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 161–168, Hsinchu, Taiwan, April 1999.

[155] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD Conference*, pages 259–270, Montréal, Canada, June 1996.

[156] J. M. Patel and D. J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *Proceedings of the eighth ACM international symposium on Advances in geographic information systems*, pages 54–61, Washington, D.C., November 2000.

[157] J. Paumard and E. Aubourg. Adjusting astronomical images using a censored Hausdorff distance. In *Proceedings of the 1997 International Conference on Image Processing (ICIP'97) - Volume 3*, pages 232–235, Washington D.C., December 1997.

[158] T. Peucker. A theory of the cartographic line. *International Yearbook of Cartography*, 16:134–143, 1976.

[159] R. Pohle and K. D. Tönnies. A new approach for model-based adaptive region growing in medical image analysis. In *CAIP '01: Proceedings of the 9th International Conference on Computer Analysis of Images and Patterns*, pages 238–246, Warsaw, Poland, September 2001.

[160] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[161] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[162] D. R. Reddy and S. Rubin. Representation of three-dimensional objects. Computer Science Technical Report CMU–CS–78–113, Carnegie-Mellon University, Pittsburgh, PA, April 1978.

[163] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 10–18, Ann Arbor, MI, April 1981.

[164] J. B. Rosenberg. Geographical data structures compared: a study of data structures supporting region queries. *IEEE Transactions on Computer-Aided Design*, 4(1):53–67, January 1985.

[165] D. Rotem. Spatial join indices. In *Proceedings of the 7th IEEE International Conference on Data Engineering*, pages 500–509, Kobe, Japan, April 1991.

[166] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference*, pages 71–79, San Jose, CA, May 1995.

[167] W. J. Rucklidge. Locating objects using the Hausdorff distance. In *Proceedings of the Fifth International Conference on Computer Vision (ICCV'95)*, pages 457–464, Boston, MA, June 1995.

[168] W. J. Rucklidge. *Efficient Visual Recognition Using the Hausdorff Distance.* Springer-Verlag, Secaucus, NJ, 1996.

[169] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

[170] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[171] H. Samet. Depth-first k-nearest neighbor finding using the MaxNearestDist estimator. In *Proceedings of the 12th International Conference on Image Analysis and Processing*, pages 486–491, Mantova, Italy, September 2003.

[172] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan-Kaufmann, San Francisco, 2006.

[173] M. Schiwietz and H.-P. Kriegel. Query processing of spatial objects: complexity versus redundancy. In D. Abel and B. C. Ooi, editors, *Advances in Spatial Databases—3rd International Symposium, SSD'93*, vol. 692 of Springer-Verlag Lecture Notes in Computer Science, pages 377–396, Singapore, June 1993.

[174] B. Seeger. Performance comparison of segment access methods implemented on top of the buddy-tree. In O. Günther and H.-J. Schek, editors, *Advances in Spatial Databases—2nd Symposium, SSD'91*, vol. 525 of Springer-Verlag Lecture Notes in Computer Science, pages 277–296, Zurich, Switzerland, August 1991.

[175] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: a dynamic index for multi-dimensional objects. In P. M. Stocker and W. Kent, editors, *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, pages 71–79, Brighton, United Kingdom, September 1987. Also University of Maryland Computer Science Technical Report TR–1795, 1987.

[176] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: a dynamic index for multi-dimensional objects. Computer Science Technical Report TR–1795, University of Maryland, College Park, MD, February 1987.

[177] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, pages 301–311, Birmingham U.K., April 1997.

[178] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. *IEEE Transactions on Knowledge and Data Engineering*, 14(1):156–171, January/February 2002.

[179] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *Proceedings of the ACM SIGMOD Conference*, pages 343–354, Dallas, TX, May 2000.

[180] H. Shin, B. Moon, and S. Lee. Adaptive and incremental processing for distance join queries. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1561–1578, 2003.

[181] J.-W. Song, K.-Y. Whang, Y.-K. Lee, M.-J. Lee, and S.-W. Kim. Spatial join processing using corner transformation. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):688–695, July/August 1999.

[182] M. Stonebraker, J. Frew, and J. Dozier. The SEQUOIA 2000 project. In D. Abel and B. C. Ooi, editors, *Advances in Spatial Databases—3rd International Symposium, SSD'93*, vol. 692 of Springer-Verlag Lecture Notes in Computer Science, pages 397–412, Singapore, June 1993.

[183] C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration for spatial selections and joins. In *Proceedings of the ACM SIGMOD Conference*, pages 455–466, San Diego, CA, June 2003.

[184] T. G. Szymansky and C. J. van Wyk. Space efficient algorithms for VLSI artwork analysis. In *Proceedings of the 20th Design Automation Conference*, pages 734–739, June 1983.

[185] K.-L. Tan, B. C. Ooi, and D. J. Abel. Exploiting spatial indexes for semijoin-based join processing in distributed spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(6):920–937, November/December 2000.

[186] M. Tasan and Z. M. Özsoyoglu. Improvements in distance-based indexing. In M. Hatzopoulos and Y. Manolopoulos, editors, *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pages 161–170, Santorini, Greece, June 2004.

[187] Y. Theodoridis, E. Stefanakis, and T. K. Sellis. Cost models for join queries in spatial databases. In *Proceedings of the 14th IEEE International Conference on Data Engineering*, pages 476–483, Orlando, FL, February 1998.

[188] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, November 1991.

[189] T. Ulrich. Loose octrees. In M. A. DeLoura, editor, *Game Programming Gems*, pages 444–453. Charles River Media, Rockland, MA, 2000.

[190] U.S. Bureau of the Census. Tiger/line files (tm). Technical report, U.S. Bureau of the Census, 1992.

[191] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 406–415, Athens, Greece, August 1997.

[192] J. van den Bercken, B. Seeger, and P. Widmayer. The bulk index join: A generic approach to processing non-equijoins. In *Proceedings of the 15th International Conference on Data Engineering*, page 257, Sydney, Austrialia, March 1999.

[193] P. van Oosterom. An R-tree based map-overlay algorithm. In *EGIS/MARI94: Fifth European Conference on Geographical Information Systems*, pages 318–327, Paris France, March 1994.

[194] P. van Oosterom and E. Claassen. Orientation insensitive indexing methods for geometric objects. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, volume 2, pages 1016–1029, Zurich, Switzerland, July 1990.

[195] J. W. van Roessel. Design of a spatial data structure using the relational normal forms. *Int. J. Geographical Information Systems*, 1(1):33–50, January 1987.

[196] J. W. van Roessel. A new approach to plane-sweep overlay: Topological structuring and line-segment classification. *Cartography and Geographic Information Systems*, 18(1):49–67, January 1991.

[197] J. W. van Roessel. An integrated point-attribute model for four types of areal GIS features. In *Proceedings of the 6th International Symposium on Spatial Data Handling*, pages 137–144, Edinburgh, Scotland, UK, September 1994.

[198] H. M. Veenhof, P. M. G. Apers, and M. A. W. Houtsma. Optimisation of spatial joins using filters. In C. A. Goble and J. A. Keane, editors, *Advances in Databases, Proceedings of 13th British National Conference on Databases (BNCOD13)*, vol. 940 of Springer-Verlag Lecture Notes in Computer Science, pages 136–154, Manchester, United Kingdom, July 1995.

[199] D. E. Vengoff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceeddings of the Goddard Conference on Mass Storage Systems and Technologies NASA Conferenece Publication 3340, Volume II*, pages 553–570, 1996.

[200] G. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxiême mémoire: Recherches sur les parallèlloèdres primitifs. Seconde partie. *Journal für die Reine und Angewandte Mathematik*, 136(2):67–181, 1909.

[201] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978.

[202] J. T.-L. Wang, X. Wang, K.-I. Lin, D. Shasha, B. A. Shapiro, and K. Zhang. Evaluating a class of distance-mapping algorithms for data mining and clustering. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 307–311, San Diego, CA, August 1999.

[203] K.-Y. Whang. The multilevel grid file—a dynamic hierarchical multidimensional file structure. In A. Makinouchi, editor, *Proceedings of the 2nd International Conference on Database Systems for Advanced Applications (DASFAA'91)*, pages 449–459, Tokyo, Japan, April 1991.

[204] D. A. White and R. Jain. Similarity indexing with the SS-tree. In S. Y. W. Su, editor, *Proceedings of the 12th IEEE International Conference on Data Engineering*, pages 516–523, New Orleans, LA, February 1996.

[205] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 68–77, Miami, FL, December 1991.

[206] C. Xia, J. Lu, B. C. Ooi, and J. Hu. Gorder: an efficient method for KNN join processing. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakely, and K. B. Schiefer, editors, *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 756–767, Toronto, Canada, September 2004.

[207] J. Xiao, Y. Zhang, X. Jia, and X. Zhou. Data declustering and cluster-ordering technique for spatial join scheduling. In K. Tanaka and S. Ghandeharizadeh, editors, *Proceedings of the 5th International Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 47–56, Kobe, Japan, November 1998.

[208] J. H. Yi, B. Bhanu, and M. Li. Target indexing in SAR images using scattering centers and the Hausdorff distance. *Pattern Recognition Letters*, 17(11):1191–1198, September 1996.

[209] X. Yi and O. C. Camps. Robust occluding contour detection using the Hausdorff distance. In *Proceedings of the IEEE Conference on Vision and Pattern Recognition (CVPR'97)*, pages 962–967, San Juan, Puerto Rico, June 1997.

[210] X. Yi and O. I. Camps. Line-based recognition using a multidimensional Hausdorff distance. *Proceedings of the IEEE Transations on Pattern Analysis and Machine Intelligence*, 21(9):901–916, September 1999.

[211] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, Austin, TX, January 1993.

[212] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-nearest-neighbors queries in spatial databases. In M. Hatzopoulos and Y. Manolopoulos, editors, *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pages 297–306, Santorini, Greece, June 2004.

[213] X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. In M. Scholl and A. Voisard, editors, *Advances in Spatial Databases—5th International Symposium, SSD'97*, vol. 1262 of Springer-Verlag Lecture Notes in Computer Science, pages 178–196, Berlin, Germany, July 1997.

[214] H. Zhu, J. Su, and O. H. Ibarra. Extending rectangle join algorithms for rectilinear polygons. In *Web-Age Information Management: First International Conference, WAIM 2000*, pages 247–258, Shanghai, China, June 2000.

[215] H. Zhu, J. Su, and O. H. Ibarra. Toward spatial joins for polygons. In *Proceedings of the 12th International Conference on Statistical and Scientific Database Management (SSDBM)*, pages 233–241, Berlin, Germany, July 2000.

[216] H. Zhu, J. Su, and O. H. Ibarra. On multi-way spatial joins with direction predicates. In *Advances in Spatial and Temporal Databases : 7th International Symposium, SSTD*, pages 217–235, Redondo Beach, CA, July 2001.

[217] M. Zhu, D. Papadias, J. Zhang, and D. L. Lee. Top-k spatial joins. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):567–579, April 2005.

[218] G. Zimbrao and J. M. de Souza. A raster approximation for processing of spatial joins. In A. Gupta, O. Shmueli, and J. Widom, editors, *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 558–569, New York, August 1998.