FAST ALGORITHMS FOR SPATIAL AND MULTIDIMENSIONAL JOINS

by

Nikos Koudas

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright by Nikos Koudas 1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
Abstract

Fast Algorithms for Spatial and Multidimensional Joins

Nikos Koudas
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
1998

Since the introduction of the relational model of data, the join operation has received much attention due to its unique feature of combining data from different relations. As Database Management Systems become richer in data types, there is increasing interest to extend the join operation to new data types, like geographical or spatial and multimedia data. Several algorithmic approaches have been proposed in the past.

In this thesis, we present new algorithms for executing join operations between new data types. The algorithms we propose are robust, and they are generic, in the sense that they can be applied to a variety of data types.

We first introduce Filter Trees, a new indexing structure, that supports efficient join operations between spatial data types. We introduce and define the basic operations for Filter Trees, analyzing their performance.

We also propose a new algorithm for joins on spatial data types, between data sets for which no indexes are available. The algorithm is suitable for the efficient processing of intermediate query results. The new algorithm, called Size Separation Spatial Join, is described and analyzed. In addition we introduce a technique called, Dynamic Spatial Bitmaps, which can improve the performance of Size Separation Spatial Join by allowing the algorithm to adapt to the specific data distributions in the data sets processed.

Finally, we propose extensions of the Size Separation Spatial Join algorithm for data sets of higher dimensionality. The new algorithm, called Multidimensional Spatial Join
(MSJ) algorithm, can efficiently perform joins between data sets containing high dimensional points.

A detailed performance comparison with algorithms previously proposed is presented in each case showing the benefits of our approach.
To the memory of my father
Στη μνημή του πατέρα μου
Acknowledgements

The years of my graduate studies have been an important period in my life up to now, mainly due to the people I have met and the ways they affected my life, knowledge and personality.

Joining the University of Maryland at College Park back in 1992, I had the chance to meet and interact with three prominent database researchers, namely Christos Faloutsos, Mike Franklin, and Nick Roussopoulos. I wish to thank Nick Roussopoulos for all the advice he gave me and for his supervision and financial support during my last semester at College Park. Christos Faloutsos was my academic supervisor at College Park. He helped me write my first paper, influenced me in many ways and helped when I much needed help. His clarity of thought, his personality and jokes inspired me in many ways. Although he might not be aware of it, he is a model for many of us. Christos, continued to provide his wisdom over the subsequent years of my studies, providing valuable comments and encouragement, following my departure from UMD. Alex Delis, my colleague at College Park, encouraged me and helped me significantly during my first steps in the graduate world. I wish also to acknowledge the help and friendship of Marios and Charlotte Camille Leventopoulos. Yiannis Papankonstantinou made sure I landed safety in the states, and I wish to thank him for all his help.

Arriving in Toronto, I had the unique opportunity to meet and have as my PhD thesis advisor, Professor Ken Sevcik. Ken, being by far the smartest person I have ever met, taught me many things over the years. His clarity of thought, sharpness and unique character were and always will be a source of inspiration. I would like to thank him for all the things he has done for me, both professionally and personally. I also wish to thank him, for tolerating my mistakes, and for all the time he devoted for me and my (often wrong) ideas. Ken was always able to understand in seconds, topics that I have been struggling for weeks or months and provide the best advise. I feel obliged to write that this thesis would not have been possible without Ken's help and supervision. His way of dealing with things, his consistency, calmness and strong personality, influenced me in many ways. I would also like to thank him for generously financing my many trips to conferences, which proved to be an educational and cultural experience.
I would like to thank the members of my PhD thesis committee, Tony Bonner, Charlie Clarke, Alberto Mendelzon and James Stewart, for attending the various checkpoints and for their comments. Rakesh Agrawal, served as my external examiner, and I wish to thank him for all his comments. Several people I met in conferences, devoted time to talk with me, comment on my work and often encourage me to continue. With the danger of forgetting someone (so I apologise in advance), these people include: Rakesh Agrawal, Chaitan Baru, Al Cameau, Christos Faloutsos, Jim Gray, H. V. Jagadish, James Hamilton, Sertrag Koshafian, Hans Peter Kriegel, Ravi Krishnamurthy, Bruce Lindsay, C. Mohan, Ken Salem, Bernie Schifer and Pat Selinger.

Generous financial support was provided by IBM corporation via a CAS student fellowship and a cooperative fellowship award. These awards also provided the unique opportunity to expose myself in an industrial environment from the early stages of my studies and shape my research ideas. I wish to thank Jacob Slonim and Kelly Lyons for all their help at IBM.

I would also like to thank Dr. H. V. Jagadish for putting his reputation on the line and being one of my references.

My friends at the University of Toronto played a big role in my studies and kept me sane over the years. I wish to thank: T.K. Anand, Stergios Anastasiadis, Melanie Baljko, Attila Barta, Michalis and Petros Faloutsos, Theo Garefalakis, Karim Harzallah, Spiros Mancoridis, Lucia Moura, Themis Palpanas, Daniel Panario, Dimitris Plexousakis, Panagiotis Tsaparas, Bil Tzerpos, Dimitra Vista, Nick Zachariadis and Danny Zilio for their friendship. The rest of the folks in our database group, George Mihaila, Davood Rafiei and David Toman provided much inspiration and thoughts during our informal database group meetings. Kathy Yen, our graduate secretary, made dealing with departmental paperwork a joyful experience. Teresa Miao provided much help with teaching assignments.

My family and friends in Greece provided emotional support since I left. I would like to thank them for being the way they were, despite the passage of time. Last but not least, I wish to thank Eleni for her smiles and for bearing my digital nature for the last 0100 years.
## Contents

1 Introduction .................................. 1
   1.1 Background ................................ 1
   1.2 Thesis Outline ............................. 2

2 Spatial Access Methods and Queries .... 4
   2.1 Spatial Queries ............................. 4
      2.1.1 Static and Dynamic Single Processor Spatial Joins Algorithms . 7
      2.1.2 Parallel Spatial Join Algorithms .................... 12
   2.2 Ordering Spatial Domains .................... 15
   2.3 Spatial Access Methods ..................... 16
      2.3.1 Quad Trees .............................. 17
      2.3.2 Grid Files ............................... 18
      2.3.3 K-D-B Trees ............................. 20
      2.3.4 The R-tree ............................. 21
   2.4 A Comparison of Spatial Access Methods ........ 23

3 Filter Trees ................................ 26
   3.1 Introduction ............................... 26
   3.2 Work Related to Filter Trees ............... 27
   3.3 Definition of Filter Trees .................. 28
      3.3.1 Assumptions ............................ 28
      3.3.2 Definition .............................. 29
      3.3.3 Hierarchy of Filters ................... 31
3.3.4 Processing Algorithms ...................................................... 33
3.4 Analysis of Filter Trees .......................................................... 40
  3.4.1 Distribution of Entities Over Levels ...................................... 40
  3.4.2 Range Query Precision and Cost ........................................... 42
3.5 Experimental Results ............................................................ 46
  3.5.1 Spatial Joins ................................................................. 47
  3.5.2 Range Queries .............................................................. 52
3.6 Conclusions about Filter Trees .................................................. 53

4 Size Separation Spatial Join ..................................................... 55
  4.1 Introduction ................................................................. 55
  4.2 Overview of Spatial Joins ..................................................... 55
    4.2.1 Partition Based Spatial Merge Joins .................................... 56
    4.2.2 Spatial Hash Joins ..................................................... 58
    4.2.3 Summary ............................................................... 59
  4.3 Size Separation Spatial Join .................................................. 60
    4.3.1 $S^3J$ Algorithm ...................................................... 60
    4.3.2 Dynamic Spatial Bitmaps for Filtering ................................ 63
  4.4 Analysis of I/O behavior ....................................................... 66
    4.4.1 Analysis of the three algorithms ..................................... 66
    4.4.2 Analytical Comparison of the Algorithms ................................ 72
  4.5 Experimental Comparison ...................................................... 74
    4.5.1 Description of Data Sets ............................................... 76
    4.5.2 Experimental Results ................................................... 77
    4.5.3 Discussion ............................................................. 86
  4.6 Conclusions about $S^3J$ ...................................................... 87

5 High Dimensional Similarity Joins ............................................ 89
  5.1 Introduction ............................................................... 89
  5.2 Problem Statement .......................................................... 90
  5.3 Survey of Various Algorithmic Approaches ................................ 91
Chapter 1

Introduction

1.1 Background

Since the late seventies an enormous amount of research and development in both academia and industry has been devoted to the efficient storage, access and processing of relational information [Cod70]. By the end of the eighties relational technology had reached maturity and many successful commercial products were based on implementations of the ideas about relational databases conceived both in academia and industrial research laboratories.

It was soon realized, however, that the relational model of data was not a panacea for business and application needs. Many new data types such as, spatial (geographical), audio, image, video and time series could not be easily represented by the model. Many researchers turned to providing extensions to the relational model, capable of incorporating, expressing and processing new interesting data types. The new systems that evolved from this extension are called Extended Relational Systems or Object Relational Database Management Systems (ORDBMS) [SM96].

This thesis proposes new algorithms for efficient processing of such data types in an Object Relational Database Management System. Specifically, the work presented in this thesis deals with extensions of the relational join operator to new data types. This operation, called spatial join, is presented, and efficient algorithms for its execution are proposed, described and analyzed. As in the case of relational joins, spatial joins deal
with the association of related information from two or more data sets. Given a user supplied predicate, the spatial join operation between two data sets will report all pairs of items from the two data sets for which the predicate is true.

The main motivation for the work in this thesis comes from the area of query processing. Every DBMS includes algorithms that implement the various operators of the model, like selects, joins, count, etc. These algorithms have to be robust, execute efficiently and offer performance guarantees for the variety of data distributions encountered in applications. The common metric adopted to assess the performance of different algorithms is the number of disk accesses required to execute an operation. Such a metric is reasonable, because most DBMS applications are constrained more by movement of data to and from disks than by processor speed. However, for an ORDBMS, the predicates executed on new data types are much more costly to evaluate and processor time might be an issue as well. The algorithms proposed in this thesis offer fast and efficient execution of spatial joins, trying to minimize both processor and disk processing costs. Moreover, from a practical point of view, we believe that they can be straightforwardly implemented as a part of an ORDBMS.

Several advanced applications can benefit from the algorithms proposed. For example, an application that deals with geographic data (Geographical Information System or GIS) can efficiently execute queries imposing a variety of predicates on geographical entities. In a similar manner, in a time series application (such as stock management) queries dealing with association of information, such as finding pairs of stocks with similar movement can benefit as well. In an image database, queries like “find pairs of images with similar percentages of a particular color”, can make use of the spatial join operation.

1.2 Thesis Outline

This thesis is organized as follows:

Chapter 2 presents related work in the area of indexing techniques for new data types. The most important spatial indexing structures and examples of their instantiations are presented. Important query types are identified, and the spatial join operation is formally
defined. Finally, this Chapter describes previous work relevant to spatial join operations.

Chapter 3 introduces Filter Trees, a new indexing structure, and describes the algorithms for manipulating and querying Filter Trees. Filter Trees are analyzed under certain assumptions about the distribution of values in the data sets they index. The Filter Tree range query and spatial join algorithms are described. Experimental results are presented showing the performance of Filter Trees for various spatial queries, including spatial joins for a variety of real and synthetic data sets.

Chapter 4 introduces the Size Separation Spatial Join algorithm and presents its analysis and variants. The Size Separation Spatial Join algorithm performs the join of data sets for which indices are not available. Thus, it is suitable for use in cases of intermediate query results. Experimental results show the performance of the Size Separation Spatial Join algorithm in comparison to algorithms previously proposed for the same problem.

Chapter 5 discusses the generalization of the Size Separation Spatial Join algorithm to more than two dimensions. In addition, several other algorithms to perform joins in high dimensional spaces are presented and discussed. This chapter contains a detailed performance study comparing the various algorithms.

Finally, chapter 6 concludes the thesis and discusses directions for future work related to Filter Trees and Size Separation Spatial Joins, as well as their multidimensional extensions. Various issues pertaining to the efficient execution of multidimensional joins on commodity parallel architectures are discussed in appendix A.
Chapter 2
Spatial Access Methods and Queries

Spatial data are data describing space and its embedded objects including points, lines, regions, rectangles, surfaces, volumes. Spatial data arise in many applications like Geographical Information Systems (GIS), and they should be supported by ORDBMS. Any query against this class of data is referred to as a spatial query. Being able to respond quickly to spatial queries requires sophisticated data organization techniques. These techniques are referred to as spatial access methods. The terms spatial data, spatial access methods and spatial queries are used in this thesis and the related literature to refer to data, access methods and queries of two and three dimensions. For problems of higher dimensionality the terms multidimensional data, multidimensional access methods and multidimensional queries are used.

The following sections provide a survey of spatial and multidimensional access methods and queries.

2.1 Spatial Queries

Given a universe of spatial objects $U$, the following (non exhaustive) list of spatial queries can be invoked:

- **Exact Match Query**: Given an Object $O$ determine if $O \in U$. 
• **Point Query:** Given a point \( P \) find all objects in \( U \) containing \( P \).

• **Range Query:** Given a \( d \) dimensional interval, \( I \), determine all objects in \( U \) having at least one point in common with \( I \).

• **Enclosure Query:** Given a \( d \) dimensional interval, \( I \), determine all objects in \( U \) enclosing \( I \).

• **Containment Query:** Given a \( d \) dimensional interval, \( I \), determine all objects in \( U \) enclosed by \( I \).

• **Nearest Neighbor Query:** Given a point \( P \) determine the object in \( U \) with the minimum distance from \( P \) according to some distance function. The distance function determines the point of each object of \( U \) to which the distance from \( P \) is computed.

• **Polygonalization:** Find all closed polygons formed by a collection of line segments.

• **Spatial Join:** Given two collections \( R \) and \( S \) of spatial objects in \( U \) and a predicate \( \theta \), find all pairs of objects \( (O_1, O_2) \in R \times S \) such that \( \theta(O_1, O_2) \) evaluates to true. Notice that \( R \) and \( S \) might be the same relation. In this case we refer to the spatial join query as a self join.

Spatial joins are the main focus of our work. The term spatial join has been used in previous work, to refer to the join operation between sets of two or three dimensional data. The term *multidimensional* or *similarity join* is used to refer to the join of higher dimensional data. In this operation, we have at least two data sets, and we would like to correlate the data in a specified way. For example, if the data sets describe lakes and cities in a state, one possible query would be “find all the cities that have a lake on their border”. In the relational domain, assuming relations of size \( n \), the worst case complexity of the join problem is \( O(n^2) \), since all tuples of one relation can possibly match with all the tuples from the joining relation in a degenerate case. Similarly to the relational join operation, a spatial join is a quadratic operation in the worst case, because the time to
produce the result is proportional to the product of the sizes of the data sets. Since the
data sets involved may be very big, algorithms are needed that yield answers quickly
in most common cases, despite the fact that the time required will be very long in the
worst case. Traditional relational join algorithms are not directly applicable in the spatial
case. Hash-based join techniques do not preserve the spatial locality needed for efficient
spatial join computation. Sort-based join techniques, which have complexity $O(n \log n)$
assuming relations of size $n$, might apply, but only after a specific spatial ordering is
specified (which is non-trivial for spatial objects).

Query processing on spatial objects is usually a two step process. Each spatial object
is approximated with its Minimum Bounding Rectangle (MBR), which is the smallest
rectangle enclosing the spatial entity. The query is first executed (applying one or more
predicates) on the MBRs, during the Filter step. The outcome of the Filter step is
checked further, during the Refinement step, applying the predicate(s) on the actual
spatial representations of the entities.

A variety of predicates may be useful in a spatial join operation. Given two entities
$E_1$ and $E_2$, predicates of interest include:

- Intersects: return true if $E_1$ and $E_2$ have at least one point in common.
- Contains: returns true if $E_1$ contains $E_2$.
- Enclosed by: returns true if $E_1$ is enclosed by $E_2$.
- Distance 'within $\varepsilon$ of': Given characteristic points in $E_1$ and $E_2$ (e.g., center of
  gravity), returns true if the characteristic points are within distance $\varepsilon$ from each
  other.

The spatial join in which the intersection (overlap) predicate is used plays a crucial role
in almost all cases. For example, for the predicates contains, enclosed by and distance
'within $\varepsilon$ of', the intersection join can be used as a filter step to identify candidate
solutions and the actual predicate can be applied to (in general) a smaller number of
candidate pairs in the refinement step.
2.1.1 Static and Dynamic Single Processor Spatial Joins Algorithms

Static Spatial Join Algorithms

In this section, we describe related previous work on static spatial join algorithms. By "static", we mean that the algorithm is applied to spatial data sets that are already represented in some type of indexing structure.

Brinkhoff et al. describe an algorithm for spatial join computation using $R^*$-trees [BKS93] (explained in section 2.3.4). In figure 2.1 we illustrate their approach. The naive spatial join algorithm between two R-trees is to perform a depth first search traversal of both indices checking for overlaps at each level. However, there is much room for improvement to this naive approach both in processor and disk access time. The first improvement in terms of processor time is to narrow the search space for intersections. Let $R$ and $S$, be two rectangles as in figure 2.1 that intersect, and let $INRS$ be their intersection. In order to join the child nodes, $RS$, and $SS$, first check if they intersect $INRS$. If yes, proceed recursively, thus following a depth first search approach; otherwise ignore them.

In addition, Brinkhoff, et al. use a plane sweep algorithm to check for rectangle intersection [PS85]. The plane sweep technique can identify rectangle intersections in time linear to the number of rectangles, thus its use incurs savings in processor time. Assume $R$ is a sequence $r_1, r_2, \ldots, r_n$ of rectangles. A rectangle $r_i$ is given by its lower left corner $(x_i^l, y_i^l)$ and its upper right corner $(x_i^r, y_i^r)$. A sequence $R$ is sorted with respect to the $X$ axis, that is, $x_i^l \leq x_{i+1}^l, 1 \leq i \leq n$. The basic idea of plane sweep, is to move a line, the sweep line, perpendicular to one of the axis, say X, from left to right. Let $R = (r_1, \ldots, r_n)$ and $S = (s_1, \ldots, s_m)$ be two sequences of rectangles. $R$ and $S$ are sorted as described above. Then the sweep line is moved to the rectangle, say $p_c$, in $R \cup S$ with the lowest $x_i$ value. If the rectangle belongs to $R$, we sequentially traverse $S$ until a rectangle is found in $S$, say $s_k$, with $x_i^k$ greater than $x_i^c$ of $p_c$. Now we know that the projections of $p_c$ and $s_j$ intersect for, $1 \leq j < k$ onto the x-axis. If their $Y$ projections also intersect, we know that rectangle $p_c$ intersects rectangles $s_j$, $1 \leq j \leq k$. If $p_c$ belongs
to $S$, we traverse $R$ analogously. Rectangle $p_c$ is marked to be processed. Then the sweep line is moved to the next unmarked rectangle in $R \cup S$ with the smallest value of $x_l$ and the same step is repeated for all unmarked rectangles. When the last entry of $R$ or $S$ is processed, all intersections have been computed.

For disk access improvement, Brinkhoff, et al. try to maximize the hit ratio in their buffer space by exploiting the spatial locality imposed by their plane sweep intersection technique and by pinning the pages in the buffer that will be used again in join processing. In addition, they propose the use of a spatial ordering technique, namely z-curves, to increase spatial locality of the objects in the buffer and thus improve the buffer hit ratio. Their experimental results indicate substantial savings both in processor and disk access time by the combination of the above techniques relative to the naive algorithm.

Huang et al. [HJR97] recently proposed an improvement to the spatial join algorithm of Brinkhoff et al. The idea is to execute a breadth first search traversal of the R-tree indices instead of a depth first search. Using this approach, a global optimization of disk reads per level is possible, thus saving more disk accesses.

Spatial join computation, based on MBR overlap, is only one step in solving the
problem. If the data are not rectangles then the MBR is only an approximation of the actual object stored on disk. Brinkhoff et al. [BKSS94] analyze the effectiveness of MBR usage in the general case of non-rectangular objects in a centralized environment. Their experiments indicate that there is much room for improvement by using alternative approximations of the objects. They propose conservative methods of approximation based on the convex hull of the object, a five point approximation [DBB83], a circle and elliptical approximations. The best performance with respect to elimination of false hits (pair of MBR's that overlap but the objects do not join) is achieved by using the convex hull approximation. However, since convex hull computation is expensive, they recommend use of a five point approximation of the object, derived using an algorithm proposed by Dori and Bassat [DBB83], resulting in a non-canonical pentagon. They also propose the use of progressive approximations such a the maximal embedded circle and the maximal embedded rectangle, based on the observation that, if the progressive approximations overlap, then the objects have to intersect.

They provide experimental evidence that, by storing these approximations together with the MBR in R-tree nodes, the false hits are reduced significantly. Finally, for the last part of the join computation, namely the join of the actual objects, they compare two techniques: (1) The use of a plane sweep technique [PS85]; (2) The use of a new structure called the $T R^*$ - tree, which is a modified R-tree that indexes the actual objects using trapezoidal approximations. They propose to store these trees on disk and retrieve those instead of the actual object, then use a MBR spatial join technique between the retrieved trees to compute the spatial join of the actual data. Although this method involves additional storage overhead and preprocessing, they provide experimental evidence that their approach pays off when the retrieved objects are large.

Orenstein [OM88] proposed a spatial join algorithm based on z-curves or Peano curves. Like Hilbert curves (explained in section 2.2), z-curves provide a mapping of a multidimensional space to one dimension. The curve is used to create an index on the spatial objects using the z-value as a key.
Dynamic Spatial Join Algorithms

In this section, we describe related work on dynamic spatial join algorithms. By “dynamic” we mean that the data sets are not indexed. This case arises when at least one data set is the result of a spatial selection or some other operation that creates an intermediate data set. For example, a new data set may be created as a result of a range query, and this set is then joined with one that is either the result of a different query or is indexed via a spatial index. As an example, consider two data sets, one containing all buildings in Toronto and another containing all parking spaces. A possible query would be to find all Government buildings that have a parking space next to them. A selection query has to be issued to find all Government buildings. A new data set is created and this data set is joined with the parking spaces data set. However, the new data set is not indexed. Lo and Ravishankar [LR94] present an algorithm to create an index “on the fly” for the derived data set. They utilize the knowledge that the new data set has to be joined with one that has already a spatial index. They use the information in the non-leaf index nodes of the indexed tree to create the so called Seeded tree. The Seeded tree is created by copying the information of the non-leaf nodes of the index and by inserting the derived data set into the new tree. The modification that they propose is to copy only the centers of the rectangles from the index to the Seeded tree. They propose to limit the disk traffic during Seeded tree construction by linking newly inserted pages in the buffer manager and write them onto disk sequentially, if necessary. Their experimental results indicate that their technique improves spatial join computation, when compared to a brute force technique (which uses the information on the derived data set to query the index) and when compared to an R-tree construction and spatial join computation [BKS93].

In subsequent work Lo and Ravishankar [LR95] present techniques to generate seeded trees from data sets on which indices to not exist, using sampling techniques. They present a series of heuristic approaches for the generation of seeded trees from base data sets along with some analysis of the buffering requirements of their proposed techniques.

Patel and DeWitt [PD96] propose an algorithm, called Partition Based Spatial Merge
Join (PBSM), to compute the spatial join of two or more datasets when indices are not available on the data sets involved in the join operation. The algorithm imposes a decomposition of the input data sets into a number of partitions. Then, after making sure that corresponding partitions fit in main memory, it proceeds to join corresponding partitions by applying main memory plane sweep algorithms [PS85].

A similar approach was independently proposed by Lo and Ravishankar [LR96] called Spatial Hash Joins. The algorithm differs from PBSM in the way partitions are formed. Utilizing techniques proposed earlier [LR95], an algorithm is developed to perform spatial joins between pairs of data sets on which indices do not exist. The algorithm is shown to perform well for several synthetic and real data sets.

Rotem [Rot93] applies the idea of a join index, proposed originally by Valduriez [Val87], to spatial data. The main idea is that, in case of joins on data that change infrequently, it is beneficial to pre-compute an index (called a join index) to facilitate the computation of the join. Rotem applies this idea to spatial joins using Grid files. He proposes the use of a plane sweep technique on the Grid directory to compute the join index and materialize it. He also gives algorithms to update the index. The use of a join index pays off only for joins in which the join selectivity is low (few objects are selected).

Gunther [Gue93] presents an analytical comparison of three spatial join algorithms, namely, nested loops, indexed and join index based. The main conclusion from the study is that the nested loops algorithm always performs the worst. Between indexed and join index based spatial joins, there is a crossover point in the performance as the selectivity of the spatial join increases. For highly selective joins (small selectivity) the use of a join index pays off. For less selective joins, an index based method is better.

A main memory indexing structure for performing spatial joins in high dimensional spaces called $\epsilon$-KDB tree was recently proposed by Shim et al. [SSA97]. The structure is a variant of KDB trees [Rob81], but the spatial join predicate (specifically a distance $\epsilon$) is used in the construction of the main memory data structure. Performance results with real and synthetic data sets, show that the structure performs well when compared with the R-tree join algorithm and Nested Loops.
2.1.2 Parallel Spatial Join Algorithms

The fact that spatial and multidimensional data sets can be extremely large has led researchers to exploit parallelism in order to execute spatial queries faster. Hoel and Samet present an algorithm for the parallel construction of R-trees on the Thinking Machines CM5 [HS93]. The algorithm assumes the presence of the complete data set in main memory. Their technique is recursive. They first associate each object with a processor (called object processors) and then they create one R-tree node associated with a designated processor (called tree processor). They compute the number of object processors that correspond to the tree processor (initially the whole data space) and recursively split the data space.

The fact that the algorithm works only for main memory data is highly restrictive since in general complete data sets cannot fit in main memory. In addition, the CM5 has not succeeded as an underlying architecture in the parallel database business.

Hoel and Samet present a spatial join algorithm for PMR quad-trees and R-trees [HS94a]. Again the algorithms assume that data sets are memory resident. In figure 2.2, we illustrate their algorithm for quad-trees. The basic idea is to create one-to-one mappings in main memory between the objects that will join in the two data sets, and initiate communication between the processors. The key point is to create the mapping only between objects that will join, thus avoiding unnecessary communication. For quad-trees, this is can be done efficiently since quad-trees impose a regular decomposition of the space. Initially both data sets are loaded into memory. Two quad trees are built in parallel, by assigning object processors to quad tree node processors. See figure 2.2 for an example of the resulting quad-tree of the data space on the left. Next, a mapping quad tree is created. The task of the mapping quad tree is to create a hierarchical regular decomposition of the resultant space (after the join), assigning processors to each quadrant. At the same time it establishes connections between the processors assigned to each quadrant of the two quad-trees already created. The process continues recursively and stops when a one-to-one mapping between the joining processors is achieved (which is always possible).
Figure 2.2: Parallel join, using quad trees on CM5

Since R-trees do not impose a regular decomposition of space such a one-to-one mapping cannot be achieved and more communication is needed. Specifically, once the two R-trees have been created in main memory (as in figure 2.3), the leaf nodes of the target R-tree are communicated to the leaf nodes of the source R-tree. Each source node creates a list of the processor indices of the target nodes with which there might be intersection and the communication array is created. Then, scheduling of the communication between the nodes takes place and spatial join operation is performed.

The above algorithms cannot be used to process disk resident data sets that are too large to fit in main memory. If these algorithms were used to handle disk resident data, the data would have to be processed in batches. Each batch would have to be a little less than half the size of main memory, and, assuming $B_1$ batches of the first data set and $B_2$ of the second, the above algorithm would have to be performed $B_1 \times B_2$ times, likely resulting in many disk accesses.

Hoel and Samet [HS94b] present a comparison of spatial join performance between R-trees and PMR-quad trees on a CM5. Again the algorithms are evaluated for main memory data sets, consisting of line segments. The results prove PMR-quad trees superior to R-trees. However, these results depend on the authors' assumptions. By assuming
main memory resident data the main disadvantage of the PMR-quad tree, namely low space utilization resulting from data replication, is no longer a problem. They use characteristic fast operations of the underlying hardware and they eliminate the redundancy through sharing, thus avoiding a lot of work which would have to be done if the data sets were disk resident.

Koudas et al. [KFK96] study the problem of declustering the leaves of an R-tree on a shared nothing parallel database architecture [DG92]. An analytical model is proposed relating the size of the declustering unit to the various parameters of the system architecture. In subsequent work, an algorithm is proposed for performing spatial joins on a shared nothing parallel database architecture using global indices [KS95].

A parallel algorithm for high dimensional spatial joins based on the epsilon-KDB tree was recently presented by Shafer et al. [SA97]. The algorithms constructs $\epsilon$-KDB trees with identical structure on each node and then proceeds to assign join pairs to nodes. Whenever a node of an epsilon-KDB tree, participates in a join pair that is assigned to two or more processor nodes, replication of the tree node takes place. After the assignment phase, each processing node performs the joins assigned to it. The performance results presented indicate that the algorithm has good performance and scalability when
compared to a space partitioning approach similar to the one used in PBSM.

Zhou et al. recently proposed an algorithm for parallel spatial join computation [ZAT97]. They follow a space partitioning approach, similar to the one used in PBSM, dividing the space into a large number of cells and mapping cells to processors. The mapping takes place with the help of a coordinator node which considers the sizes of cells and makes sure load balancing is achieved among the nodes. Each spatial entity overlapping cell boundaries is replicated.

2.2 Ordering Spatial Domains

In a one dimensional domain, a unique total order exists through the use of a primary key. In a spatial or multidimensional domain, however, creating a total order is a more complex problem. The reason is that, no unique total order exists, and there are many alternatives to choose among in mapping points in a multidimensional space onto a line.

Two main techniques have been proposed. The first chooses a characteristic point to describe each spatial element and provides a total order based on this characteristic point. For example, given a set of vectors \( \mathbf{v}_i, 1 \leq i \leq N \), of dimension \( d \), where \( \mathbf{v}_i = v_{i1}, \ldots, v_{id} \), one might choose \( v_{i1} \) and provide a total order of the set. Certain parameters can influence the choice of the characteristic point, such as clustering in the resulting space.

Another approach proposed in the literature is the use of space filling curves. A space filling curve is a function, which maps an \( d \) dimensional space to a one dimensional space. Several such functions exist.

The first space filling curve we describe is the Hilbert curve. A Hilbert curve is shown in figure 2.4. It visits all discrete points in a \( k \)-dimensional grid exactly once while never crossing itself. Thus, it can be used to linearize the points of a grid. The basic curve on a \( 2 \times 2 \) grid, denoted by \( H_1 \), is shown in figure 2.4. To derive a curve of order \( i \), each vertex of the basic curve is replaced by the curve of order \( i - 1 \), after appropriate rotation and/or reflection. Figure 2.4 also shows the Hilbert curves of order 2 and 3. When the order of the curve tends to infinity, the resulting curve is a fractal with fractal dimension 2. Hilbert curves as well as algorithms to construct them are presented by
Faloutsos and Roseman [FR89]. Another kind of space filling curve is the z-curve [Ore86]. Faloutsos and Roseman present algorithms to construct z-curves [FR89]. Jagadish [Jag90] presents an analytical study of the performance of curve-based techniques for clustering. An analytical comparison of the clustering properties of various space filling curves is presented by Moon et al. [MJFS97].

2.3 Spatial Access Methods

Significant work on the topic of spatial access methods has been done for the last 20 years [Sam90]. Spatial access methods fall into two categories. Those that organize the embedding space that contain the data and those that organize the data items themselves. In the first category belong two prominent methods, namely the Quad-tree [Sam90] (and all its variants) and the Grid file [NHS84]. In the second category belong two other prominent methods, namely the K-D-B tree [Rob81] and the R-tree [Gut84]. For convenience we will discuss the methods in the special case of two dimensions, although all the methods can be generalized to three or more dimensions.
2.3.1 Quad Trees

The basic characteristic of the Quad-tree is that it imposes a regular decomposition of space up to a specific resolution. The original Quad-tree was a direct generalization of binary search trees and thus not a space partitioning technique. It was soon adapted as a spatial access method and evolved to a space partitioning basis. The root node represents the entire space. Each node in the tree has four descendents (labeled NW, NE, SW, SE), representing the four quadrants of the region represented by the node.

There are many variants of Quad trees proposed in the literature. Each variant is tailored to a specific kind of data. Variants exist for representing lines (the PM Quad tree family [Sam90]), rectangles (the MX-CIF Quad tree family [Sam90]), volume data and curvilinear data [Sam90]. Quad-tree variants proposed for line and rectangular data have problems. Due to the regular decomposition of the space it is difficult to represent overlapping segments or rectangles. Existing solutions sacrifice space utilization and also complicate insertion and deletion.

For representing and querying lines, the PMR quad-tree [NS86] has been shown to have good performance. PMR quad-trees allow representation of overlapping and non-overlapping data. They handle overlapping data by using random splits of an overflowing node only once, thus paying a price since they require variable size nodes. Space utilization in the worst case can be very bad, but the designers claim that, for practical data sets, where overlapping lines are few, this structure is very efficient.

Quad-trees (especially point quad-trees, which represent points) have been used extensively in graphics and vision, due to their property of regular decomposition up to a specific resolution (pixels). They support a wide variety of spatial queries, including range search, point queries and nearest neighbor queries. Quad-trees guarantee good worst case complexity for searches, $O(2N^{\frac{1}{2}})$ for $N$ two dimensional data points. This result can be generalized to $k$ dimensions in which case it becomes $O(kN^{\frac{1}{k}})$. However their worst case space requirement is exponential in the number of data points.

Figure 2.5(a)(b) shows a point data set and the corresponding quad tree representation. Gray index nodes indicate the existence of data in the path below. White index
nodes, indicate that the quadrant below is empty.

### 2.3.2 Grid Files

A space partitioning approach is taken by the Grid file. The designers had two objectives in mind: Efficient processing of range queries in large linearly ordered domains and performing point queries in exactly two disk accesses. The method can be used to index records with a large number of attributes, but as an example, consider records with two attributes. The values of the record attributes can be used to represent the record as a point in the two-dimensional space. In order to be able to represent all the possible attribute value pairs, each axis is partitioned into a number of cells equal to the cardinality of the domain of each attribute. Such a scheme can be implemented with a two-dimensional array. If the record is present, the corresponding array cell provides a pointer to it. Otherwise, the corresponding position will be empty. However this simplistic approach is highly space inefficient. This is because we preallocate space in the array to reference all the possible records with two attributes.

The approach taken by the Grid file, for two dimensional data spaces, is to define two structures, (\(k\) in the \(k\)-dimensional case) called *linear scales* specifying an initial partition...
of the attribute space in each dimension. These scales are relatively small and can be memory resident. The scales define a Grid directory (that is disk resident) and are used to index into it. Assuming a point query on point \((X, Y)\) the linear scales provide the disk address of the corresponding directory block. A disk access is made and the directory block is brought into memory. Then the directory block provides the disk address of the bucket (disk page) that contains the specified record, if any. An example is shown in figure 2.6(a)(b). The data space is partitioned using the linear scales. The directory is disk resident. Using the linear scales, we can access the appropriate directory block on disk. One more disk access is required to get the actual data. We assume disk pages and directory blocks of capacity two. A search for point C proceeds as follows: Using the linear scales the coordinates of C are checked and a block address is calculated using a mapping function. The mapping function can be very simple, for example, it can be based on the fact that C's X coordinate falls in the first partition of the horizontal linear scale (the same for C's Y coordinate). A disk access for the corresponding directory block, gives the address of the page on which the point is located.

Since directory occupancy may be low, the Grid file allows many directory cells to point to the same disk page (instead of allowing only a one-to-one mapping of directory
cells to disk pages, which results in low space utilization). In a dynamic environment, insertions and deletions force buckets to split, thus triggering adjustment to directory pointers and linear scales. A method is suggested for keeping track of the split history of buckets that eases merging policy decisions. One problem with Grid files is that, when the attributes are very strongly correlated, the directory structure grows rapidly, resulting in bad space utilization. The BANG file [Fre87] tries to overcome this problem by allowing a nested directory block structure.

Grid files inspired much subsequent work. They have been used as the basis of declustering algorithms and load balancing algorithms in parallel data bases [GD90].

2.3.3 K-D-B Trees

K-D-B trees [Rob81] impose an irregular and not unique decomposition of space. There are two types of nodes in a K-D-B tree: region nodes, and point nodes. Region nodes contain a set of pairs (region, node-id) with region corresponding to the bounding rectangle of the space represented by the node pointed to by node-id. Point nodes contain actual data or pointers to actual data. K-D-B trees provide fast answers to range queries on point spaces. Figure 2.7(a)(b) presents a data space and its corresponding K-D-B tree representation, assuming the insertion sequence was, A, B, C, D, E and the bucket (page) size is two. In this example, two bucket splits occur. We assume that the first split is based on the X coordinate and the second on the Y. There are many algorithms for splitting a bucket. Cyclic splitting based on all coordinates in a round-robin fashion gives good results [Rob81]. However for highly correlated data or when knowledge is available of what kinds of queries will be asked and how frequently, other splitting patterns will be more effective.

The original description of K-D-B trees left unspecified several important aspects of the structure. It did not specify the organization of a region node or how the split of each node would take place, in the case that decomposition based on one dimension is not possible. The hB-tree [LS90] generalized the K-D-B tree structure solving these problems. The authors propose the use of a k-d tree for the organization of region nodes. In addition they proposed that, during a node split, more that one dimension might
Figure 2.7: (a) Sample data space. (b) $K$-$D$-$B$ tree after insertion of $A,B,C,D,E$, using cyclic splitting.

participate. The resulting structure is shown to have good average case performance and space utilization.

### 2.3.4 The R-tree

The R-tree [Gut84] is another extension of B-trees for multidimensional data. A geometric object is represented by its minimum bounding rectangle (MBR). The index organizes the space as a tree data structure where non-leaf nodes contain entries of the form $(ptr, R)$ where $ptr$ is a pointer to a child node in the R-tree; $R$ is the MBR that covers all the rectangles in the child node. Leaf nodes contain entries of the form $(obj\_id, R)$ where $obj\_id$ is a pointer to the object description, and $R$ is the MBR of the object. The main innovation in the R-tree is that parent nodes are allowed to overlap. This way, the R-tree can guarantee good space utilization and remain balanced. Figure 2.8 illustrates data rectangles (in black) organized into an R-tree with fanout three. Figure 2.9 shows the hierarchical structure of the same R-tree, where nodes correspond to disk pages.

The R-tree inspired much subsequent work, the main focus of which was to improve the search time. Sellis et al., proposed a structure called, $R^+$-tree [SRF87]. This structure
Figure 2.8: Data (dark rectangles) organized in an R-tree

Figure 2.9: The file structure for the R-tree of the previous figure (fanout = 3)
avoids the overlap between non-leaf nodes of the tree by clipping data rectangles that cross node boundaries. Subsequently an improved structure called $R^*$-tree was proposed [BKSS90], and it was shown experimentally to perform better than its predecessors. The main idea is the concept of forced reinsert, which is an analog to the deferred splitting in B-trees. When a node overflows, some of its children are carefully chosen, and they are deleted and re-inserted, usually resulting in a better structured R-tree.

There has been some work on packing for R-trees. Packing refers to the action of putting together (clustering) in the same page spatial data that are close together in the underlying space. The underlying motivation is that data with spatial proximity are frequently processed together. Hence putting them in the same page will result in fewer disk accesses. In addition, if the R-tree leaves are packed, MBR's are small, so the probability that they overlap is reduced, resulting in shorter search time. Different algorithms for packing have been proposed; using the lower $x$ coordinate of a rectangle for sorting [RL85], using z-curves [Ore86], and Hilbert curves [FR89]. Faloutsos and Roseman show that Hilbert curves outperform all the previously proposed methods for clustering R-trees [FR89]. Recently, a new R-tree based on Hilbert curves was presented along with its algorithms for the basic spatial operations [KF94].

2.4 A Comparison of Spatial Access Methods

Having described different techniques to index spatial data, a question one might ask is how these methods perform when compared to one another. Various methods have been demonstrated to be best in various specific situations. It’s not clear which would be best on average when applied to many real data sets.

Several parameters influence the quality of indexing provided by different access methods:

- The kind of data to be indexed; Some structures perform well for one kind of data (e.g., points) while others do not.

- The kind of queries of interest to the application; Some queries are very easy to
answer with a specific access methods while others are harder. Query workload is an important factor when considering the choice of an access method.

- The space required by the access method; The classical space/time tradeoff seems to have great importance. Sacrificing space utilization can improve search time and vice-versa.

A sound, general performance comparison of the methods is very difficult. Care must be taken in order to compare the structures and their concepts rather than comparing specific implementations. A comparison depends both on implementation details and workload assumptions. In addition, the data sets used in an experimental evaluation might influence (or bias) the results. Any comparison should contain a mixture of synthetic as well as real data sets. The synthetic data sets previously used include uniformly distributed data sets, Zipfian data sets [FK94], Gaussian data sets, and data sets that are generated according to some fractal law [FK94]. One measure proposed to characterize synthetically generated data sets is the coverage coefficient defined as:

\[
\text{Coverage Coefficient} = \frac{\sum_{\text{All spatial objects in data space}} \text{Area of spatial object}}{\text{Area of the data space}}
\] (2.1)

Although several attempts have been made to define a benchmark specifying a set of queries of interest and also propose a collection of publicly available data sets to be used in performance studies, unfortunately these efforts did not succeed [KSSS89][SFGM93].

Hoel and Samet report their results from a qualitative study [HS92]. They compare PMR-quad trees, $R^*$-trees and a hybrid structure based on $R^+$-trees. Their comparison is based on five queries, namely, point query, nearest neighbor, polygonization, and two kinds of range queries. The main result of this study is that no clear winner can be found. As far as space utilization is concerned, Quad-trees were the worst and $R^*$ the best. PMR-trees were most efficient in terms of query execution time for some queries, but $R^*$ or $R^+$ were best for others.

The R-tree family can achieve space utilization close to 100%. However, as space utilization increases, insertion/deletion performance deteriorates. Since R-trees approximate spatial objects with MBR's and all spatial objects can be approximated this way, it
is sufficient for a DBMS to support only R-trees as a spatial access method. The Quad-
tree family appears to be very customizable. Quad tree structures have been proposed
for many kinds of spatial data. All of them try to control space utilization and guarantee
good search performance.

K-D-B trees (based on a limited amount of reported research) appear to be suitable
for point data and to have satisfactory space utilization.

Grid files can guarantee good performance, once a good partitioning of the linear
scales is found. Any point query can be answered with only two disk accesses. However,
when the attributes are highly correlated, in theory, the Grid file suffers a directory size
increase and range query performance is not very good. Experimental work has not been
reported however, showing this extreme. If statistical information for the data space is
available, which can lead to careful partitioning of the space, Grid files are a good choice.

The factors that influence the choice among indexing methods are:

- knowledge of the expected workload of queries;
- knowledge of the characteristics of the data space;
- knowledge of the application’s performance requirements.

All those factors should be considered in order to make a good decision.
Chapter 3

Filter Trees

3.1 Introduction

We introduce a new file structure called Filter Trees. We describe algorithms for the construction of Filter Trees and for the processing of range queries and spatial joins on Filter Trees. We demonstrate, using a mixture of analysis and experimentation with a prototype implementation, that Filter Trees have substantial performance advantages over previously proposed file structures in processing some spatial queries of the types needed in applications such as those mentioned earlier, specifically spatial joins.

Filter Trees derive their relative advantages through the principles of hierarchical representation, size separation, and locality of accesses. Filter Trees involve a recursive binary partitioning of the data space in each dimension. Entities associated with a particular level are all grouped together. Each entity is placed at the lowest-level of the tree at which it is fully enclosed by a single cell of the partition at that level. This method of determining the level at which an entity is stored tends to cause larger entities to be stored high in the tree (because they can be contained only in large cells), while smaller entities tend to sink to lower levels of the tree because they fit into smaller cells. Sometimes small entities will be caught at higher levels in the tree because they happen to lie across the boundary between two large cells. However, under reasonable statistical assumptions about where entities are placed, the fraction of such entities is small.

The algorithms for processing Filter Trees are designed to limit the portion of the
index and data space that must be explored in order to respond to a query, and also to maximize the degree of locality within the portion of space that is explored. The locality is exploited by using space filling Hilbert curves (of different degrees at different levels of the hierarchy) to order the items stored at a particular level.

### 3.2 Work Related to Filter Trees

A primary goal for Object Relational Data Base Management Systems (ORDBMS) [Sto93] and Geographical Information Systems (GIS) is to provide efficient access to data that describe two dimensional or three dimensional objects.

Abel and Smith [AS83] first proposed a method to organize rectangles based on their sizes. They used a concept similar to the locational code of linear-quadtrees [Gar82] and organized the rectangles in a B-tree index. A similar approach was proposed by Kedem [Ked82] [Sam90]. Six and Widmeyer used size separation to extend Grid files to represent hyper-rectangles rather than points [SW88]. With the multilayer Grid file structure, it is possible to avoid the necessity of clipping or redundantly storing entities. Hutflesz, Six, and Widmeyer later proposed the R-File, which uses a multi-resolution representation to yield improved performance for range queries relative to R-trees. They show that the R-File does better than R-trees whenever the coverage coefficient is high (greater than about 1.2) [HSW90]. The R-file also uses a space filling curve (specifically, z-ordering) to achieve locality of access. Another form of size separation was proposed by Guenther [Gue91]. The inclusion of oversize shelves in hierarchical spatial data structures allowed for the storage of items that would otherwise span many nodes at lower levels of the hierarchies. This helped to avoid over-fragmentation of entities.

Orenstein and Manola proposed PROBE for use in image database applications [OM88]. The method uses a hierarchical representation reflecting the containment of objects in sub-quadrangles. Z-ordering is also used to assure a degree of locality. The method is shown to be useful in evaluating range queries. Its effectiveness in processing spatial joins is less well explored. The required algorithms are described, but no experimental performance results are reported. It appears that no guarantees can be made
about how much buffer space is sufficient to attain a given level of performance, or how few times it is necessary to read each block.

Thus, hierarchical representation, size separation, and space-filling curves have all been used in various previously proposed multidimensional information structures. However, by combining them, Filter Trees can perform spatial joins with a guaranteed minimal number of block reads from disk. Other methods that have been proposed cannot make such guarantees, and, except for R-Trees, experimental results on the performance of spatial joins of hyper-rectangles are not available.

3.3 Definition of Filter Trees

3.3.1 Assumptions

In this section, we formally define Filter Trees. We introduce Filter Trees in a rather limited context initially to facilitate the presentation.

The two dimensional case is easiest to understand and is most relevant to geographic information systems for representing maps. However, there are many other applications that require use of three or more dimensions. Fortunately, the two basic mechanisms of the Filter Tree, namely binary recursive partitioning and Hilbert curve ordering, both generalize to higher dimensions [Jag90]. Because the number of cells at level \( j \) in a \( k \) dimensional Filter Tree is \( k^j \), however, the number of levels that can practically be used decreases as \( k \) increases.

When the domains of the attributes that define the multidimensional space are not \([0,1]\) and/or the distributions of the attribute values are known to be non-uniform, transformation of the attribute values is required before the Filter Tree is constructed. Spatial objects in two-space can be mapped into the unit square, so that the coordinates of the center of their Minimum Bounding Rectangles are located inside the unit square. This means that some vertices of the MBR can be outside of the unit square. Assume \( F_x(x) \) is an approximation (which is continuous and invertible) to the cumulative distribution function (CDF) of the minimum and maximum coordinate values in a given dimension
for all the entities. Then the inverse mapping,

\[ x = F_{x}^{-1}(x') \]  \hspace{1cm} (3.1)

where \( x' \) is a coordinate of the point in the original space, can be used to make the transformed coordinate, \( x \), approximately uniformly distributed on the unit interval.

If the attribute values in the two dimensions are independent, then the transformation will cause entities to be mapped uniformly over the unit square. If they are not independent and the correlation is known, then the dependence can be taken into account in the inverse mapping of one of the two coordinate values in order to once again approach an approximately uniform distribution of entities over the unit square after the transformation.

While many (perhaps most) spatial database applications deal with static or nearly static sets of spatial entities, there are other spatial database applications in which updates reflect the addition, deletion, movement, and transformation of spatial entities. These updates occur interleaved with queries through the lifetime of application. To handle such applications with Filter Trees, some space can be systematically left in each block to allow for efficient insertions and modifications to the set of spatial entities. As has been shown with variations of B-trees, this technique can lead to storage utilization in the 80% to 90% range and still handle updates efficiently. Correspondingly, the processing of range queries and spatial joins will generally require 10% to 20% more block transfers than in the static case in which blocks are fully packed.

### 3.3.2 Definition

In two-dimensional space, we assume that each entity to be stored in the database consists of (1) a shape, which is defined by a simple, closed polygon, and (2) additional information. The entity records, which include the shape and the additional information, are stored in blocks to form the bulk of the database. It is desirable (although not mandatory) to cluster the records so that those contained in one block tend to be located close to one another or even overlap in two-space.
From the two-dimensional shape of each object, we may calculate the minimum bounding rectangle (MBR), which is the smallest rectangle that is aligned with the axes of the two-dimensional space and encloses the entity's shape. The storage of and access to an entity in the Filter Tree is based completely on its MBR. For convenience of exposition, we will refer to the two dimensions as $x$ and $y$, although their interpretation in specific cases will depend on the application.

The minimum bounding rectangle is specified by the coordinates of its lower left corner $(x_l, y_l)$ and upper right corner $(x_h, y_h)$, where $x_l$, $x_h$ (respectively, $y_l$ and $y_h$) are smallest and largest values of the $x$ (respectively $y$) coordinate, anywhere along the perimeter of the entity's shape. The coordinates of the centre of the MBR are $(x_c, y_c)$, where $x_c = \frac{x_l + x_h}{2}$, and $y_c = \frac{y_l + y_h}{2}$.

Physical storage of both MBRs and entity records requires a serialized ordering of the entities. To obtain this serialized order while retaining locality of overlapping and neighboring entities in two-dimensional space, we map the center of each entity's MBR to a space filling Hilbert curve. The Hilbert curve value of the center of the MBR $(x_c, y_c)$ can be calculated from the binary representations of the coordinates, $x_c$ and $y_c$. For $z$ curves, this requires only an interleaving of the bits of the binary representations of $x_c$ and $y_c$; for Hilbert curves in two dimensions, the transformation is more complex, involving manipulation of bit pairs based on a state transition table. The algorithms and state transition table are available elsewhere [Bia69][SK95].

Our use of Hilbert curves involves relating $(x, y)$ coordinate pairs in the unit square (with $k$ bits of precision) to Hilbert values in the unit line (with $2k$ bits of precision)\footnote{Most previous work has enumerated degree $k$ Hilbert curves using the integers 0 to $4^k - 1$. The binary fractions used in our work are precisely the integers used by others divided by $4^k$ for degree $k$ Hilbert curves.}. Thus each of the $4^k$ cells in level $k$ of a Filter Tree can be identified either by a pair of $k$-bit $x$ and $y$ coordinates, or equivalently by the corresponding $2k$ bit binary fraction representing a Hilbert value.

Figure 3.1 illustrates this relationship for Hilbert curves of degree 1 and 2. Note that:

- The $2k$ bit Hilbert value of a level $k$ cell is the prefix of the $2k+2$ bit Hilbert values of the four level $k+1$ subcells, where the four subcells are distinguished by
appending 00, 01, 10 and 11 as the least significant bits of the $2k+2$ bit binary fractions.

- For a cell at level $k$ with a $2k$ bit Hilbert curve value, the cell at level $k-1$ containing the level $k$ cell corresponds to the Hilbert value of the subcell, but truncated after the $2k$-2nd bit.

- For every $2k$ bit binary fraction, $b$, the corresponding cell is adjacent to the two cells that correspond to $b + 4^{-k}$ and $b - 4^{-k}$ (unless one of these numbers is outside $(0,1)$).

The last point means that the Hilbert curve is an optimal space-filling curve in the sense that no serialization of the cells can do any better than having every pair of adjacent $2k$ bit binary fractions correspond to cells that are adjacent in two-space.

### 3.3.3 Hierarchy of Filters

The Filter Tree is based on a hierarchy of regular grids that divide the unit square into subsquares. At level $j$, the grid consists of lines at $\frac{k}{2^j}$, $k = 0, \ldots, 2^j$ in both the $x$ and $y$ dimensions. For example, the level 3 grid partitions the unit square into 64 squares of size $1/8 \times 1/8$. The hierarchy has $L$ levels, where the smallest MBR's have sides no smaller than $2^{-L}$.

Each entity to be stored in the Filter Tree is associated with a level in the tree by examining its MBR. At an intuitive level, we drop the MBR through the grids at the levels of the hierarchy. The MBR of an entity comes to rest at the first level at which its
MBR is not fully contained within a single cell. If an MBR has one side of length greater than \(2^{-j}\), then it will be associated with a level no lower than \(j\). Thus, relatively large rectangles are guaranteed to be associated with higher levels in the tree, and relatively small rectangles will tend to be associated with lower levels. According to their locations, however, some small rectangles will be associated with high levels (because they happen to straddle grid lines at high levels).

More mathematically, the level of the hierarchy with which an entity is associated is determined as follows: Express the \(x\) and \(y\) coordinates of the MBR as binary fractions, and count the number of initial bits in which \(x_l\) agrees with \(x_H\) and also \(y_l\) agrees with \(y_H\). If that number is \(j\), then the entity is associated with level \(j\) of the hierarchy.

Figure 3.2 illustrates this process for three rectangles of differing sizes. Entity A is large and resides at level 1 of the Filter Tree. Entity B is much smaller, and fits within a 1/8 by 1/8 cell, so it is associated with level 3 of the tree. Entity C is smaller still, but its location on the line \(x = 1/2\) causes it to be associated with level 0 of the tree.

The bulk of the data in a Filter Tree is located in the entity records. Each entity record contains all the information associated with the corresponding entity. There is some advantage to ordering the entity records according to the Hilbert values of the centers of their MBRs so that proximity in two-space is preserved in the serialized entity record file as much as possible.
Entity records are located through entity descriptors. For each entity, there is a corresponding entity descriptor stored in the entity descriptor file. An entity descriptor contains:

- specification of the minimum bounding rectangle (MBR) of the entity, \((x_l, y_l), (x_h, y_h)\),
- the Hilbert curve coordinate associated with the center of the MBR, \(H(x_c, y_c)\),
- a pointer to the disk block in which the corresponding entity record is stored.

The entity descriptor file is organized so that: (1) the descriptors for all the entities associated with a particular level are stored together; and (2) within each level, the descriptors are ordered by the Hilbert value of the centre of their MBRs. A consequence of (2) is that the entities contained in a particular cell of a particular level will all be stored contiguously. The descriptors are packed into blocks, with each block containing about 50 to 100 entity descriptors (assuming 32 bytes per descriptor and a block size in the range of a few kilobytes).

For the part of the entity descriptor file associated with each level of the Filter Tree, there is a cell index. The cell index is a B-tree that records the Hilbert value of the last entity descriptor in each block. This requires one entry in the cell index for each block of the entity descriptor file for the level (plus a small additional cost for the upper levels of the B-tree). The entry has size 12 bytes (8 bytes to store the Hilbert value in maximum precision, and 4 bytes to store its position in the file).

### 3.3.4 Processing Algorithms

#### Construction Algorithm

The algorithm for constructing a Filter Tree from a set of entity records is given in Figure 3.3. In the static case, the effort to construct the tree (sorting the entity records themselves into Hilbert order, packing them into blocks, and storing the blocks contiguously on secondary storage) will be amortized over all the queries answered using the tree. Filter Trees, however, are a dynamic data structure. Insertions, deletions and modifications can be handled by applying the standard B-tree algorithms for the corresponding
Given a set of entity records:

- Create from each entity record an entity descriptor:
  1. From the shape of the entity, derive its minimum bounding rectangle.
  2. From the corners of the MBR, determine the level of the Filter Tree at which the entity is to be stored.
  3. From the coordinates of the center of the MBR, derive the Hilbert value associated with the entity.
  4. Include in the entity descriptor a pointer to the block of secondary storage that contains the entity record.

- Create the entity descriptor files for each level of the Filter Tree:
  1. Group the entity descriptors for each level of the tree.
  2. For each level, sort the entity descriptors according to Hilbert value order.
  3. Pack the entity descriptors into contiguous blocks of secondary storage, inserting an entry for each block into the B-tree cell index for the level. The entry specifies the Hilbert value of the last entity descriptor in the block.

Figure 3.3: Filter Tree Construction Algorithm

operation in the level file to which the entity belongs.

Spatial Joins Using Filter Trees

In this section, we describe how spatial joins are executed using a Filter Tree structure. Spatial joins deal with correlations of entities between two or more spatial data sets according to some correlation predicate. This predicate can specify conditions on the overlap between two entities, the maximum (minimum) distance between them, etc., and only entity pairs that satisfy the predicate will be included in the output of the spatial join. This algorithm is useful when the predicates applied lead to blocks of manageable
Spatial joins find many applications in GIS and they are particularly useful in spatial data mining applications [NH94]. More specifically, spatial joins are useful in identifying spatial relationships between objects in a data set.

Join processing proceeds in two steps. The first step, called the filter step, identifies a list of candidate pairs that might qualify to be in the output. The next step, called the refinement step, tests the full predicate against the full entity records for each object pair produced during the filter step. Two different indexing methods will identify the same set of candidate pairs and will transfer the same number of entity records from disk, applying the same algorithms for predicate evaluation between them. Therefore, the critical factor in choosing a method for performing spatial joins is the performance of the filter step.

A spatial join between two Filter Trees involves an index sweeping process. However, the structure of the Filter Tree makes the sweeping process very efficient. For any pair of data sets, their full spatial join can be computed with the minimal amount of I/O, namely by reading each block of the entity descriptor file at most once. In addition, no other algorithm can read fewer blocks from the files than the Filter Tree algorithm in order to perform the join.

Consider the hierarchies of filters, $F_1$ and $F_2$, shown in figure 3.4. There are four levels in each hierarchy, normalized in the same subspace. If we wish to search for matches between entity descriptors in cell 0 of $F_1$ and all the cells of $F_2$ we may restrict our search to cell 0 of $F_2$ and its enclosing cells at higher levels (in the direction of the arrow in figure 3.4). No other cells need be considered, since, by the definition of the Filter Tree hierarchy, cells are disjoint. In a similar fashion, matching the descriptors in the 15th cell of level 2 in $F_1$ involves looking at the corresponding cell in $F_2$ and its enclosing cell at level 1 only.

The spatial join algorithm is designed to allow every cell at each level of the tree to be processed in this way while reading each block of the descriptor index file only once. This is accomplished by sweeping through the entity descriptor files at each level of each participating Filter Tree in increasing Hilbert value order, bottom up.

We identify processing intervals within the range $(0, 1)$ in terms of end markers taken
from each block of the entity descriptor file. Let $e_{ij}^F$ be the highest Hilbert value of any entity descriptor in the $j$th block of level $l$ of the Filter Tree $F$. There are as many end markers as there are blocks in the entity descriptor files of both trees together. We sort the full set of $e_{ij}^F$ values and delete any duplicates. Then the Hilbert value ranges delineated by successive pairs of end marker values in the sorted list, have the property that they are fully contained within one block at each level of each participating tree. Consequently, it is possible to process each such interval in turn while keeping in memory just one block from each level of each tree. When processing of all possible join pairs has progressed to Hilbert value $e_{ij}^{F_i}$, then we are done with the $j$th block of level $l$ of tree $F_i$, and we replace it by the $j + 1$st block of level $l$, enabling us to proceed with the next processing interval. All $e_{ij}^{F_i}$ values are not necessarily unique. Processing intervals ended by non-unique values will simply cause more than one block to be replaced before starting the next processing interval.

Within a processing interval, the following actions are carried out. Levels 0 to $L$ of each tree are addressed in turn, and the spatial join step illustrated in Figure 3.4 is carried
out on the set of entities in the current block of that level. Let $S_l^{F_i}(e_n, e_{n+1})$ denote the set of entities in level $l$ of tree $F_i$ that have Hilbert values in the range $(e_n, e_{n+1})$. By the way the processing intervals were defined, all the entity descriptors in all these sets will be in memory while the processing interval is treated. Then for levels $l = 0, \ldots, L$ in turn, we:

- match entities in $S_l^{F_1}(e_n, e_{n+1})$ against those in $S_{l-i}^{F_2}(e_n, e_{n+1})$ for $i = 0, \ldots, l$.
- match entities in $S_l^{F_2}(e_n, e_{n+1})$ against those in $S_{l-i}^{F_1}(e_n, e_{n+1})$ for $i = 1, \ldots, l$.

(Note that the ranges of $i$ differ in the two steps in order to avoid matching $S_l^{F_1}(e_n, e_{n+1})$ and $S_l^{F_2}(e_n, e_{n+1})$ twice.)

The spatial join carried out in this way is as efficient as possible, reading each entity descriptor block only once, and yet requiring that only one block of each level of each tree be in memory at a time (except in exceptional circumstances where a large number of entities have identical coordinates). By doubling the (small) memory requirement, a double-buffering technique can be used to overlap the reading of the one block at a particular level with the processing of the previous one.

**Range Queries**

In this section, we present the range query algorithm for Filter Trees. Given a query window specified by its lower left and upper right point coordinates, we wish to retrieve all entities in the tree that overlap this window. Denote the coordinates of the lower left point by $(x_l, y_l)$ and the coordinates of the upper right point by $(x_h, y_h)$. In order to answer the query, we have to search each level in the Filter Tree. However, searching within each level can be very efficient, because we can identify the blocks that needed to be examined.

At each level, each cell that covers any part of the query area must be examined. Within each level, the set of cells to be examined will form a set of Hilbert value intervals. The union of the intervals at level $k$ will be a subset of the intervals at level $k-1$, reflecting the fact that some cells included at level $k-1$ have only one or two (rather than four) subcells included at level $k$. 
Once an interval to be scanned is identified, the cell index can be used to identify the first and last blocks of the entity descriptor file that contain entities with Hilbert values in the interval. Then all blocks from the first through the last can be read with a single I/O request.

At lower levels of the tree (say 10 and below), the number of cells is so large that we must avoid having to enumerate all the cells in a range query. (Most cells at these levels will be empty anyway, since the number of cells will surpass the number of entities stored in the tree.) Because the construction algorithm for Filter Trees packs the contents of successive cells (empty and otherwise) into blocks, we need only determine what sequence of blocks contain entities with Hilbert values in a specified range.

In order to determine a set of Hilbert value intervals that together cover all the cells touched by a range query, the following approach can be used. Choose a specific level of the Filter Tree, called \( c \), to be the \textit{containment level} for processing the query. This means that intervals to be processed will be identified and expressed with Hilbert values of precision \( 2c \) bits.

From the query coordinates, \((x_l, y_l), (x_h, y_h)\), we can identify the minimal rectangular set of cells at level \( c \) that completely covers the query area. Every interval that passes through the query area starts and ends with one of the cells on the outer border of the rectangular area. Consequently, we can identify all the relevant intervals by traversing the perimeter of the rectangular area, and keeping track of all level \( c \) cells that are the start and/or the end of an interval. For each level \( c \) cell on the border of the query area, the Hilbert values of the cell and its neighboring cell outside the query area are calculated and compared. (Cells covering the corners of the query will have to be compared with adjacent cells in each dimension.) When the Hilbert value of the border cell is exactly \( 4^{-c} \) bigger than that of its neighbor, then that cell is the start of a new interval; when the Hilbert value of the border cell is exactly \( 4^{-c} \) smaller than that of its neighbor, the border cell is the end of an interval. By recording all the cells that start intervals and all those that end intervals while traversing the entire query border, and then simply sorting the two sets, all intervals are identified by pairs of entries in the two sorted sets. Choosing a larger value for \( c \) causes the intervals selected to include less marginal area
outside the query at the cost of having a larger number of border cells to traverse.

Section 3.4.2 will present an analysis of the effect of choosing a particular containment level. The appropriate choice depends more on the number of entities stored in the Filter Tree than on the precise dimensions of the query. For around 10,000 entities, level 6 is a good choice, whereas for 10,000,000 entities level 11 is good. The analysis to support these choices is given in Section 3.4.2

Each interval determined by the steps outlined above can be used to identify a sequence of blocks in the entity descriptor file for each level of the tree. Each sequence of blocks can be read with a single bulk I/O request [Wei89][SLM93]. If $k$ is the lowest level of the Filter Tree at which the query area is fully enclosed in a single cell, then only a single interval (or sequence of blocks) will be required at levels 0 through $k$ of the tree. Below level $k$, there will generally be two or more intervals involved, each corresponding to a sequence of blocks. These sequences of blocks may be adjoining or even overlapping in a single block at the ends. By considering all the sequences of blocks involved for a particular level of the filter tree and merging all sequences that overlap or are adjacent, it is possible to do a minimal number of bulk I/O requests to obtain all the entities relevant to the query at that level of the Filter Tree. Note that it may pay to merge two sequences even if they are separated by a block or two rather than adjacent or overlapping, since the single longer bulk I/O request including the intervening blocks may be less costly than two I/O requests for the sequences separately.

**Other queries**

Refering back to section 2.1, several spatial queries, besides range searches and spatial joins, are of interest. The exact match, enclosure and containment query are special cases of the range query. They can be answered by Filter Trees, by adding some more predicates in the result of the appropriate Filter Tree range query.
3.4 Analysis of Filter Trees

In this section, we analyze some properties of Filter Trees. For this purpose, we will make specific assumptions about the distributions of sizes and placements of (the minimum bounding rectangles of) entities stored in the Filter Tree.

3.4.1 Distribution of Entities Over Levels

First, we consider a Filter Tree of $d \times d$ objects, assuming that the object centers are uniformly distributed over the unit square. As was pointed out in section 3.3.1 some coordinates of the object's MBR can be outside the unit square. We seek to identify the probability distribution of these objects across levels of the Filter Tree. At Filter Tree level $j$, $d \times d$ objects will fall through only if their centers are at least distance $\frac{j}{2}$ from the lines $\frac{i}{2j}$ for $i = 0, 1 \ldots 2j$ in both the $x$ and $y$ dimensions. Thus, in order to fall through level $j$, the center of a $d \times d$ object must be in one of $4^j$ squares, each of which has area $(\frac{1}{2})^2 - d^2$. Consequently, the fraction of $d \times d$ objects that fall below level $j$ is

$$f_d(j) = 4^j(\frac{1}{2^j} - d)^2 = (1 - 2^j d)^2$$

Since the fraction that fall through level $j-1$ is $(1 - 2^{j-1} d)^2$, then the fraction that reside precisely level $j$ is

$$f_d(j) = (1 - 2^j d + 4^{j-1} d^2) - (1 - 2^{j+1} d + 4^j d^2) = 2^j d(1 - \frac{3}{4} 2^j d)$$

(3.2)

Knowing that the cumulative total at levels 0 through $j$ is $2^{j+1} d - 4^j d^2$, we can conclude that the distribution of level occupancy for $d \times d$ objects is:

$$f_d(j) = \begin{cases} 
  d(2 - d) & j = 0 \\
  2^j d(1 - \frac{3}{4} 2^j d) & j = 1, \ldots, k(d) - 1 \\
  (1 - \frac{1}{2} 2^{k(d)} d^2) & j = k(d)
\end{cases}$$

(3.3)

where $k(d) = \lceil -\log_2 d \rceil$ is the lowest level to which any $d \times d$ object can fall (since $d$ must be less than $2^{-k}$). Then the average level occupied in the Filter Tree by $d \times d$ objects is:

$$I(d) = \sum_{j=0}^{k(d)} j f_d(j)$$

(3.4)
Table 3.1 illustrates the distribution $f_j(d)$ for various values of $d$. Note that, for the internal levels of the tree, since $2^j/d = 2^{j+1}/2$, $f_d(j) = f_2d(j+1)$. If we restrict our attention to only those square objects that are located entirely inside the unit square then the fractions of objects per level in table 3.1 are different. They can be derived from table 3.1 by dropping level 0 and normalizing the fractions at the other levels such that the probabilities in each column sum to one.

The distribution of level occupancy and the average level occupied by squares of size $d$ both indicate that less that 25% of the squares reside at levels higher that the three possible ones. This demonstrates the size separation achieved by the Filter Tree structure.

If the probability density function of the sizes of objects to be stored in the Filter Tree is $p(d)$, then the aggregate distribution of level occupancy is given by:

$$f(j) = \int_0^\infty p(t)f_t(j)dt$$  \hspace{1cm} (3.5)

There are many possible assumptions that can be made about the distribution $p(d)$. 

<table>
<thead>
<tr>
<th>Level</th>
<th>$d=1$</th>
<th>$d=1/2$</th>
<th>$d=1/4$</th>
<th>$d=1/8$</th>
<th>$d=1/16$</th>
<th>$d=1/32$</th>
<th>$d=1/64$</th>
<th>$d=1/128$</th>
<th>$d=1/256$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0.75</td>
<td>0.4375</td>
<td>0.2344</td>
<td>0.1211</td>
<td>0.0615</td>
<td>0.0310</td>
<td>0.0156</td>
<td>0.0078</td>
</tr>
<tr>
<td>1</td>
<td>0.25</td>
<td>0.3125</td>
<td>0.2031</td>
<td>0.1133</td>
<td>0.0596</td>
<td>0.0305</td>
<td>0.0154</td>
<td>0.0078</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.2500</td>
<td>0.3125</td>
<td>0.2031</td>
<td>0.1133</td>
<td>0.0596</td>
<td>0.0305</td>
<td>0.0154</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.2500</td>
<td>0.3125</td>
<td>0.2031</td>
<td>0.1133</td>
<td>0.0596</td>
<td>0.0305</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.2500</td>
<td>0.3125</td>
<td>0.2031</td>
<td>0.1133</td>
<td>0.0596</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0.2500</td>
<td>0.3125</td>
<td>0.2031</td>
<td>0.1133</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>0.2500</td>
<td>0.3125</td>
<td>0.2031</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>0.2500</td>
<td>0.3125</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.2500</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I(d)$</td>
<td>0.25</td>
<td>0.8125</td>
<td>1.5781</td>
<td>2.4570</td>
<td>3.3955</td>
<td>4.3645</td>
<td>5.4389</td>
<td>6.3411</td>
<td></td>
</tr>
</tbody>
</table>
One of interest is that the average coverage of the unit square is the same for objects of all allowed sizes (that is, \( \frac{p(d')}{d'^2} = \frac{p(d'')}{d''^2} \) for any \( d', d'' \) pair). If the bounds on object sizes are \( d_{\min} \) and \( d_{\max} \), then

\[
p(d) = \frac{1}{d^2} \left( \frac{1}{d_{\min}} - \frac{1}{d_{\max}} \right) \quad (3.6)
\]

For this case,

\[
f(j) = \frac{\int_{d_{\min}}^{d_{\max}} L(t) \, dt}{\int_{d_{\min}}^{d_{\max}} \left( \frac{1}{d_{\min}} - \frac{1}{d_{\max}} \right) \, dt} \quad (3.7)
\]

Some of our experiments (presented in Section 3.5) are based on this distribution.

The analysis above can be generalized to apply to rectangular entities rather than square ones. For rectangles of size \( d_1 \times d_2 \), the fraction that fall below level \( j \) is given by

\[
4^j \left( \frac{1}{2^j} - d_1 \right) \left( \frac{1}{2^j} - d_2 \right) = 1 - 2^j (d_1 + d_2) + 2^{2j} d_1 d_2
\]

(3.8)

Since the fraction that fall through level \( j - 1 \) is \( 1 - 2^{j-1} (d_1 + d_2) + 2^{2(j-1)} d_1 d_2 \), the fraction that reside at level \( j \) is

\[
f_{d_1 d_2}(j) = 2^{j-1} (d_1 + d_2) - \frac{3}{4} 2^{2j} d_1 d_2
\]

(3.9)

So the overall distribution is given by:

\[
f_{d_1 d_2}(j) = \begin{cases} d_1 + d_2 - d_1 d_2 & j = 0 \\ 2^{j-1} (d_1 + d_2) - \frac{3}{4} 2^{2j} d_1 d_2 & j = 1, \ldots, k(d) - 1 \\ 1 - 2^{k-1} (d_1 + d_2) + 2^{2(k-1)} d_1 d_2 & j = k(d) \end{cases}
\]

(3.10)

where \( k(d) = \lceil -\log_2(\max(d_1, d_2)) \rceil \). The average level at which \( d_1 \times d_2 \) rectangles reside can be determined as before.

### 3.4.2 Range Query Precision and Cost

In describing the algorithm for processing range queries in Section 3.3.4, we pointed out the importance of limiting the total length of the Hilbert value ranges that are searched to process the range query. Here we analyze the tradeoff between the computation invested to restrict the ranges and the excess portion of space searched outside the query area.
At each level of the Filter Tree, we must examine each cell that is either enclosed or intersected by the border of the query range. For lower levels of the tree, however, there are too many cells to consider each one individually. Instead, we choose (carefully) a particular level of the Filter Tree to be the containment level, $c$, and calculate the minimal set of Hilbert value ranges required to cover all cells at that level that are contained in or overlap the query range.

Consider now a particular range query with dimensions $d_x$ by $d_y$, and a chosen containment level, $c$. Figure 3.5 illustrates the situation that must hold whenever $\min(d_x, d_y) > 2^{-c+1}$. The range query processing algorithm described in Section 3.3.4 identifies and scans all the Hilbert value ranges that cover the $n_x \times n_y$ cells at level $c$. The cost of identifying the ranges is the calculation of the Hilbert value for each boundary cell and their external neighboring cells. This requires a total of $4(n_x + n_y - 1)$ calculations of a Hilbert value from $(x, y)$ coordinate pairs.

The portion of Hilbert value ranges searched unnecessarily (because it is outside the query area but inside the bordering cells) is $n_x n_y 2^{-2c} - d_x d_y$. Allowing for the worst possible dimensions, $d_x$ and $d_y$, and the worst possible alignment of the query with the
cells at level $c$, an upper bound on the portion of Hilbert values scanned unnecessarily, $W$, is given by:

$$W = n_x n_y 2^{-2c} - d_x d_y \leq (d_x 2^c + 2)(d_y 2^c + 2)2^{-2c} - d_x d_y = 2^{-(c-1)}(d_x + d_y + 2^{-(c-1)}) \quad (3.11)$$

since $d_x \geq (n_x - 2)2^{-c}$ and $d_y \geq (n_y - 2)2^{-c}$. Expressing the number of required Hilbert value calculations, $n_H$, in terms of $d_x$ and $d_y$, we have:

$$n_H = 4(n_x + n_y - 1) \leq 4((d_x + d_y)2^c + 3) \quad (3.12)$$

For a range query of dimensions $d_x \times d_y$ on a Filter Tree that stores $N$ spatial entities, we would like the containment level, $c$, to provide an appropriate tradeoff between the computation required ($n_H$) and the excess area scanned ($W$). The fact that increasing $c$ by one roughly halves $W$ while roughly doubling $n_H$ suggests that any weighted sum of $W$ and $n_H$ will have a concave upward shape indicating the existence of an optimal $c$ value that minimizes the function. Further, the optimal $c$ value will be one for which the two components of the cost function have approximately equal magnitude. In particular consider minimizing the cost function:

$$C_{\text{total}} = (n_H \times c \times C_H) + (W \times \frac{N}{b} \times C_B) \quad (3.13)$$

where:

- $C_H = \text{processor time required per level to convert } (x,y) \text{ to a Hilbert value},$
- $C_B = \text{cost of reading and scanning a block of entity descriptors},$ and
- $b = \text{blocking factor of entity descriptors}.$

The first term represents the cost of doing all the Hilbert value calculations of cells along the border of the query, and the second term estimates the cost of processing Hilbert value ranges outside the query if the intervals to be scanned are chosen at level $c$. Thus, the best choice of $c$ is the one for which:

$$n_H \times c \times C_H \approx W \times \frac{N}{b} \times C_B \quad (3.14)$$
Table 3.2: Best values of $c$ for different database sizes

<table>
<thead>
<tr>
<th>$N$</th>
<th>best $c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>6 or 7</td>
</tr>
<tr>
<td>100,000</td>
<td>8</td>
</tr>
<tr>
<td>1,000,000</td>
<td>9 or 10</td>
</tr>
<tr>
<td>10,000,000</td>
<td>11</td>
</tr>
</tbody>
</table>

or

$$4((d_x + d_y)2^c + 3) \times c \times C_H \approx 2^{-(c-1)}(d_x + d_y + 2^{-(c-1)}) \frac{N}{b} \times C_B \quad (3.15)$$

Retaining only dominant terms on each side and dividing by $(d_x + d_y)$ yields:

$$c \times 2^{c+2} \times C_H \approx \frac{N}{b} \times 2^{-c} \times C_B \quad (3.16)$$

or

$$c \times 2^{2c+1} \approx \frac{N \times C_B}{b \times C_H} \quad (3.17)$$

In our implementation $b$ is about 60 and $c \times C_H$ is very close to $c \times 0.05$ milliseconds. If we assume $C_B$ is about 30 milliseconds then the equation becomes $c \times 2^{2c+1} \approx 10 \times N$.

Based on these assumed parameter values, Table 3.2 shows the best choice of $c$ for various values of $N$.

By retaining only the dominant terms of equation 3.15 in equation 3.16, the dependence of choice of $c$ on the query dimensions is lost. While in general we can afford a slightly larger value of $c$ for queries with smaller dimensions (because their perimeters are smaller), this is a secondary effect. It is sufficient to choose $c$ once for each Filter Tree according to the number of entities it contains.

The right hand side of equation 3.15 provides a conservative estimate of the excess cost of retrieving blocks that contain no entries relevant to the query. Depending on how the entity descriptors are packed into blocks, there may be fewer blocks composed completely of entities outside the query than is suggested by the estimate in the right hand side of equation 3.15. This means that smaller values of $c$ may suffice to limit
the number of blocks accessed as much as possible. Thus, the values in Table 2 can be interpreted as the largest values of \( c \) that would pay off if the distribution of actual entities led to the best possible packing of entity descriptors into blocks.

### 3.5 Experimental Results

In order to assess the performance benefits and limitations of Filter Trees, we conducted a series of experiments involving spatial joins and range queries, using the algorithms described in Section 3.3.4. We experimented with both real and synthetic data sets. Our Filter Tree implementation was written in C, under the SunOS operating system, and it consists of approximately 2000 lines of code.

The Filter Trees in our experiments have at most 21 levels (numbered 0 to 20), because the real data sets that were available to us produce at most 21 levels in their Filter Tree representation. We used data sets extracted from the TIGER data file of US Bureau of the Census [Bur91]. The first one consisted of 53,145 line segments representing road segments from Long Beach County, California. We will refer to this set as the LB data set. The second file consisted of 39,068 line segments representing road segments in Montgomery County, Maryland. We will refer to this file as the MG data set. We used these data sets because they have been used previously by other researchers. While they are "real", it is unlikely they are "typical" of spatial data sets because they treat sequences of highway segments and hence exhibit a low degree of overlap among intervals. For that reason, we also generated some synthetic data sets using various discrete probability distributions.

In a Filter Tree, the distribution of the sizes of the entities is of great importance, since it determines the occupancy of each level. We experimented with two distributions to generate synthetic data sets:

- **equal area coverage** (equation 3.7). This is the distribution used in our analysis in Section 3.4. We generated one data set following this distribution, having 50,000

---

2All of our experiments were conducted on a Sun Sparc 20 with a 60MHz Supersparc+, SPEC Int 92 4492, SPEC FP 92 4888.
descriptors in levels 5 to 12. We refer to this as the EA data set.

- triangular shaped distribution. Given a "peak" level and the min and max levels, the sizes of MBRs in the synthetic data set has a triangular shaped distribution.

More formally the "triangular distribution" is defined as:

\[
p(d) = \begin{cases} 
    x_1 + \frac{d-x_1}{x_2-x_1}(\frac{x_2-x_3}{x_3-x_1}) & x_1 \leq d \leq x_2 \\
    x_3 - \frac{x_3-d}{x_3-x_2}(\frac{x_2-x_3}{x_3-x_1}) & x_2 \leq d \leq x_3 
\end{cases}
\] (3.18)

where \(x_1, x_3, x_2\) correspond to the minimum, maximum and peak level. The motivation for using the triangular distribution came from observation of the distributions of the sizes in the LB and MG data sets. Using the "triangular distribution", we generated two synthetic data sets with 50,000 descriptors: Set \(TR_1\) was generated using \(x_1 = 4, x_2 = 17, x_3 = 20\) and \(TR_2\) using \(x_1 = 4, x_2 = 15, x_3 = 20\).

For all the experiments we conducted with Filter Trees, we present the corresponding performance of Hilbert R-trees for comparison. The experiments are based on the static versions of both Filter Trees and R-Trees. We chose to present performance numbers for Hilbert R-trees because they outperform all other variants in the R-tree family for range queries, and we expect they are also better for spatial joins due to their clustering properties [KF94]. It would be desirable to compare the performance of Filter Trees against other spatial data structures, but others either focus on range queries rather than spatial joins [HSW90], or experimental performance results for them are not available [OM88].

### 3.5.1 Spatial Joins

We present and discuss the experimental results obtained from the application of our spatial join algorithm to the real and synthetic data sets. For all experiments, we present the measured response time and the proportions of I/O and CPU time. The estimates for I/O time are obtained by precisely counting the number of I/O operations occurring during each experiment and charging 30 ms for each I/O operation. CPU time is then

---

3We chose to restrict the size of the Filter Tree for this data set, due to the practical problem of generating the distribution of equation 3.7 with 50,000 descriptors over 21 levels.
the measured response time minus the estimated I/O time. Although these estimates are not exact, they suggest the balance between I/O and CPU time in the join algorithms. Leaf index blocks have exactly the same structure for both access methods. However the index fanout of the Filter Tree is much higher than that for R-trees. For our prototype implementation the fanout for Filter Trees was 63 and for R-trees 42 (~ 34% higher for Filter Trees). In an enhanced implementation, Filter Trees can have fanout up to three times bigger than that of R-trees.

For comparison with the Filter Tree join algorithm, we implemented the best R-tree join algorithm proposed by Brinkhoff et al. [BKS93]. When the indexes have the same height, the algorithm proceeds top-down sweeping index blocks at the same height. At a specific height, the pairs of overlapping descriptors are computed and, at the same time, the rectangles of their intersections are computed also. This information is used to guide the search in the lower levels, since descriptors not overlapping the rectangle of intersection of their parents need not be considered for the join. The algorithm uses a buffer pinning technique that follows a greedy approach trying to keep relevant blocks in the buffer in order to minimize block re-reads. When the indexes do not have the same height, the algorithm proceeds as described above up to a certain point and then degenerates into a series of range queries.

For all the experiments, we assumed that the R-tree indexes and the Filter Tree cell indexes fit entirely in main memory. This is a realistic assumption even for large data files, and it is especially true for Filter Trees since the index size is smaller than for R-trees for most data sets. For our spatial join experiments, we addressed the following types of joins: (a) self joins (which are useful in identifying pairs of overlaps within a data set) and (b) joining two distinct data sets. For the latter, we used two different alternatives:

- joining one of $LB$ and $MG$ with one of $TR_1$ and $TR_2$, or

- joining a data set ($D$) and a synthetic data set ($D'$) generated from $D$ as follows: If $x_{max}$ and $y_{max}$ are largest sizes of any entity in $D$ in the $x$ and $y$ dimensions, respectively, then for each entity in $D$, we generate a new entity in $D'$ having as
a lower left point the center of the entity from \( D \) and sizes in \( x \) and \( y \) uniformly distributed between zero and \( x_{max} \) and \( y_{max} \) respectively. That way, a synthetic set with statistical properties similar to \( D \) is generated.

![Graphs showing performance of self joins for real and synthetic data sets.](image)

- **(a) LB data set, self join**
- **(b) EA data set, self join**

Figure 3.6: Performance of self joins for real and synthetic data sets. The percentage presented on the x-axis, corresponds to the amount of buffering available to the operation, as a percentage of the total file size.

In figure 3.6a, we present the performance of self joins for the LB data set, for R-trees and Filter Trees. For the R-tree join, we varied the buffer size available during the join operation and we present it as a percentage of the total number of blocks of both files. Increased buffer size improves the I/O behavior of the R-tree join algorithm. This basically means that the buffer hit ratio increases, since more blocks can stay memory resident. The buffer pinning part of the R-tree join algorithm tries to minimize the number of re-reads for data blocks and the increased buffer size obviously helps.

The LB data set in its Filter Tree representation has 19 levels. This means that the
Filter Tree join can proceed with only 38 blocks of buffer space, which is only 2.2% of the total set of blocks. In general, the Filter Tree join algorithm requires buffer space, equal to the sum of the numbers of levels of the Filter Trees on the data sets involved. Filter Trees provide 10% savings in response time when 5% buffering is available for R-trees. The Filter Tree performance is matched by the R-tree when 20% buffering of the underlying space is provided to the R-tree. Figure 3.6b presents the results of the same experiment using the EA data set. Filter Trees can perform the join with almost 50% savings in response time with 2.1% buffer space, relative to an R-tree with 5% buffering. Even with 20% buffering available for R-trees, Filter Trees still achieve 23% savings requiring only 2.1% buffer space.

(a) Join performance results for $MG$ and $TR_1$  (b) Join performance results for $MG$ and $MG'$

Figure 3.7: Join performance for R-trees and Filter trees using the MG data set

Figures 3.7a,b present experimental results for the join performance of Filter Trees and R-trees, using the $MG$ data. Filter Trees perform the best in both cases achieving 32% and 23% savings in response time respectively relative to the R-tree with 5% buffering
Figures 3.8a,b present join results for the $TR_1$ and $TR_2$ data sets with sets $TR'_1$ and $TR'_2$. The general trends for the performance of the R-tree join algorithm remain the same, with increased buffer size improving the total response time. However, for these data sets, the buffer pinning mechanism of the R-tree algorithm is not so effective, since a higher buffering percentage is needed in order for the algorithm to attain I/O efficiency. In particular, even with 20% buffering, R-trees have to read each block three times on average to perform the join. The Filter Tree join algorithm can proceed with only 34 blocks, which is 2.1% of the total file size. Comparing figures 3.8a and 3.8b, it is interesting to note that, as the peak of the distribution is shifted toward lower levels, the R-tree join algorithm becomes less efficient. An increase in the number of larger entities in the file causes more ambiguity in the R-tree index. As a consequence, the I/O and
CPU time requirements of the R-tree join algorithm is higher\textsuperscript{4}. We expect that, in the scope of real life spatial data base applications, the performance benefits of the Filter Tree approach will range somewhere between those reported for the \emph{LB} and \emph{MG} data sets (in figure 3.6) and those for the \emph{TR1} and \emph{TR2} data sets (in figure 3.8).

### 3.5.2 Range Queries

In this section, we present experimental results for the performance of range queries on Filter Trees. Filter Trees, due to their size separation principle, require at least one disk access at every level of the tree in order to answer a range query. We present the total number of blocks transferred versus query sizes for one real data set (\emph{LB}) and one synthetic data set (\emph{TR1}).

![Figure 3.9: Range Query Performance for Real and Synthetic Data sets](image)

We processed 100 random queries inside \emph{LB} and \emph{TR1} and we present the average number of disk accesses per query. Figure 3.9a presents the results of two experiments. For the first experiment, buffering was turned off. In figure 3.9a, we can see that R-trees perform better for range searches on \emph{LB}. For small queries (on the order of 0.001 of the space) R-trees perform an average of 3-4 disk accesses to answer the queries. The same

\textsuperscript{4}The above observations hold for an additional distribution we experimented with, in which the descriptor fraction at each level follows the Zipf distribution.
queries, in Filter Trees, require one disk access for each level and incur a higher cost. As the query size increases, both R-trees and Filter Trees require more disk accesses on the average. For the second experiment shown in figure 3.9a, we provided 5% buffering of the total file size for both R-trees and Filter Trees. For R-trees no improvement is observed. This is expected since the queries are random. However for Filter Trees, the average cost of each disk access is lower for any query size, because the cost of visiting each level in the hierarchy is amortized over all queries. For Filter Trees, lower levels are likely to fit in a single block (as it is the case for levels 2 and 3 of the \( LB \) set). These levels, as well as other levels with few blocks per level, will remain in memory, as each query will use them. Consequently, the cost of accessing these level is amortized over all queries.

Figure 3.9b presents range query performance results for \( TR_1 \). For this set, R-tree searches are not very efficient. Small queries require on average almost 20 disk accesses. For this data set, Filter Trees are able to perform a little better for range searches, since they need about one disk access per level (\( TR_1 \) has 17 levels). When the degree of overlap between MBRs becomes larger and the index height increases, R-tree searches become inefficient, because the search follows many paths down to the leaves and often finds nothing relevant. Filter Trees can adapt better to distributions with high overlap between MBRs. Their performance for large range queries remains worse than R-trees however.

We have presented separate algorithms for (1) spatial joins of pairs of Filter Trees, and (2) range queries on a single Filter Tree. In fact, however, range queries can be considered as a special case of spatial join in which one entity set contains only a single entity. Our spatial join algorithm thus can be used to answer a range query, but, because it makes no special provisions to do so, the performance is not as good as that for the special range query algorithm. In fact, the performance is sensitive to both the size of the range query and the relative position of the query in the unit square. A small query that happens to fall early in the linearization of the space will be processed with little additional cost, but one that falls near the end of the linearization would cause almost the entire index file of the Filter Tree to be read unnecessarily.
3.6 Conclusions about Filter Trees

We have presented Filter Trees, an efficient structure for performing spatial join operations between sets of spatial objects. The Filter Tree structure is based on three principles:

- **Hierarchical Representation** – Each entity is associated with a level that corresponds to a particular granularity of space partitioning.

- **Size Separation** – Entities of different sizes tend to be associated with different levels of the tree.

- **Spatial Locality** – Within each level, entities are ordered by their positions along a space-filling Hilbert curve in order to cause entities in a portion of the multidimensional space to map to contiguous portions of the linear storage space as much as possible.

Together these principles lead to a file structure that is capable of supporting spatial joins more efficiently than alternatives that have been proposed and evaluated previously. The experimental results described in Section 3.5 provided evidence of the benefits of using Filter Trees when spatial joins are frequent. For synthetic databases in which the spatial entities are randomly positioned, Filter Trees proved to require much less I/O for spatial joins relative to R-trees, but they required significantly more I/O for responding to range queries. Consequently, the choice between Filter Trees and R-trees depends on the frequency of spatial joins in the workload. Space partitioning, as provided by Filter Trees and PMR-Quadtrees [NS86], is a good approach when spatial join operations are involved between sets of line segments or sets of approximated spatial objects. This fact is supported both by our results for Filter Trees and the experimental results provided by Hoel and Samet [HS95].

For the LB and MG data sets, the advantage of Filter Trees relative to R-trees for doing spatial joins was much reduced. It is possible that this is due to the special structure of those databases where the entities stored are MBR's of successive highway segments, which by definition tend to not overlap extensively. We believe that many
real spatial databases that store two dimensional objects (as opposed to the more one dimensional highway segments in the LB and MG data sets) will lead to performance in between the extremes seen in our experiments. Evidence for this is provided by our experiments with spatial joins of the MG data set with synthetic sets, where savings of 23% and 32% were observed for the two join strategies we investigated.
Chapter 4

Size Separation Spatial Join

4.1 Introduction

We introduce a new algorithm to perform the Spatial Join (SJ) of two or more spatial data sets for which indices are not available. Our algorithm, named Size Separation Spatial Join (S3J), is a generalization of the relational Sort Merge Join algorithm. S3J is designed so that no replication of the spatial entities is necessary, whereas previous approaches have required replication. The algorithm does not rely on statistical information from the data sets involved to efficiently perform the join. We introduce and describe the algorithm, analyze its I/O behavior, and compare it with the behavior of previous approaches. Using a combination of analysis and experimentation with an implementation, we demonstrate the performance benefits of the new algorithm.

4.2 Overview of Spatial Joins

We consider spatial data sets that are composed of representations of points, lines, and regions. The shapes of spatial objects are rarely regular. In order to facilitate indexing and query processing, spatial objects are usually described by their Minimum Bounding Rectangle (MBR) or some other approximation [BKSS94].

Two algorithms have been proposed recently to solve this problem for the case where the data sets do not fit in main memory. Patel and DeWitt [PD96] introduced Partition Based Spatial Merge Join (PBSM) to compute the spatial join of two data sets without the use of indices. Lo and Ravishankar [LR96] also presented an algorithm called Spatial
Hash Joins for the same problem. In the following subsections, we describe these two algorithms in greater detail.

### 4.2.1 Partition Based Spatial Merge Joins

Partition Based Spatial Merge Join (PBSM) is a generalization of the sort merge join algorithm. Given two spatial data sets, \( A \) and \( B \), the algorithm uses a formula to compute a number of partitions into which to divide the data space. These partitions act as buckets in hash joins. Once they are filled with data, only corresponding partitions for the two data sets must be processed to locate all candidate joining pairs. However, since the entities in the two data sets are in general not uniformly distributed, the number of objects that fall in various partitions will vary. To improve the chances of achieving balanced partition sizes, the algorithm partitions the space into a larger number of tiles and maps the tiles to partitions, either round robin or using a hash function.

A spatial entity might intersect two or more partitions. The algorithm requires replication of the entity in all the partitions it intersects. Once the first spatial data set has been partitioned, the algorithm proceeds to partition the second data set, using the
Given two spatial data sets, A and B, and the number of tiles,

- Compute the number of partitions
- For each data set:
  1. Scan the data set;
  2. For each entity, determine all the partitions to which the entity belongs and record the entity in each such partition.
- Join all pairs of corresponding partitions (repartitioning, if necessary).
- Sort the matching pairs and eliminate duplicates

Figure 4.2: The PBSM Algorithm

same number and placement of tiles and the same tile to partition mapping function. Depending on the predicate of the spatial join, it might be the case that, during the partitioning of the second data set, a spatial entity that does not overlap with any tile can be eliminated from further processing since it cannot possibly join with any entities from the first data set. We refer to this feature of PBSM as filtering. Figure 4.1a presents a tiled space with three objects. Assuming four partitions, one possible tile-to-partition mapping is \((A, B, E, F)\) to the first partition, \((C, D, G, H)\) to the second, \((I, J, M, N)\) to the third and \((K, L, O, P)\) to the fourth. Under this scheme object Obj1 will be replicated in the first and second partitions.

Once the partitions are formed for both spatial data sets, the algorithm proceeds to perform the join on partition pairs and writes the results to an output file. If the partition pairs formed do not fit in main memory in entirely, they are repartitioned using the same technique. Corresponding partitions are loaded in main memory and a plane sweep technique is used to evaluate the predicate. Since partitions may include some replicated objects, the algorithm has to detect (via hash or sort) and remove duplicates before reporting the candidate joining pairs. The complete algorithm is summarized in figure 4.2.
When both spatial data sets involved in the join are base sets and not intermediate results, one can adaptively determine the number of tiles one should use in order to achieve good load balance. For intermediate results, however, the appropriate number of tiles to use is difficult to choose, since statistical information is not available and an adaptive technique cannot be applied. If an inappropriate number of tiles is used, the algorithm still works correctly; however, using too few tiles may result in high load imbalance resulting in a lot of repartitioning, while using too many may result in an excessive number of replicated objects. Note that replication takes place in both data sets. The amount of replication that takes place depends on the characteristics of the underlying data sets, the number of tiles, and the tile to partition mapping function.

4.2.2 Spatial Hash Joins

Lo and Ravishankar [LR96] proposed Spatial Hash Joins (SHJ) in order to compute the spatial join of two (or more) unindexed spatial data sets. The algorithm starts by computing the number of partitions \(^1\) into which the data space should be divided. The computation uses a formula proposed by the same authors in earlier work [LRSS]. Once the number of partitions is computed, the first data set is sampled. The centers of the spatial objects obtained from sampling are used to initialize the partitions. Then the first data set is scanned and the spatial entities are assigned to partitions based on the nearest center heuristic [LR95]. Each spatial entity is placed in the partition for which the distance from its center to the center of the partition is minimum. Once an entity is inserted in a partition, the MBR of the partition is expanded to contain the entity if necessary. When the MBR of the partition is expanded, the position of its center is changed. At the end of this process, the partitions for the first data set are formed. Notice that no replication takes place in the first data set.

The algorithm proceeds by scanning the second data set and partitioning it using the same partitions as adjusted to accommodate the first data set. If an entity overlaps multiple partitions, it is recorded in all of them, so replication of spatial entities takes

\(^1\)The authors use the term slot [LR96], but in order to unify terminology and facilitate the presentation, we use the term partitions.
Given two spatial data sets $A$ and $B$,

- Compute the number of partitions
- Sample data set $A$ and initialize the partitions
- Scan data set $A$ and populate partitions, adjusting partition boundaries
- Scan data set $B$ and populate partitions for $B$ using the partitions of $A$ and replicating where necessary.
- Join all pairs of corresponding partitions

Figure 4.3: The SHJ Algorithm

place at this point. Any entity that does not overlap with any partition can be eliminated from further processing. Consequently filtering can take place in this step of the algorithm. Figure 4.1b presents one possible coverage of the space by partitions after the partitioning of the first data set. In this case, object $Obj_1$ of the second data set will have to be replicated in partitions $A$, $B$ and $C$ and object $Obj_3$ in partitions $C$ and $D$.

After the objects of the second data set have been associated with partitions, the algorithm proceeds to join pairs of corresponding partitions. It reads one partition into main memory, builds an R-tree index on it, and processes the second partition by probing the index with each entity. If memory space is exhausted during the R-tree building phase, LRU replacement is used as outer objects are probed against the tree. The complete algorithm is summarized in figure 4.3.

4.2.3 Summary

Both PBSM and SHJ divide the data space into partitions, either regularly (PBSM) or irregularly (SHJ) and proceed to join partition pairs. They both introduce replication of the entities in partitions in order to compute the join. Replication is needed to ensure that joining pairs are not missed in the join phase when entities cross partition boundaries. When data distributions are such that little replication is introduced during the
partition phase, the efficiency of the algorithms is not affected. However, for other data distributions, replication can be unacceptably high, and can lead to deterioration of performance. Prompted by the above observation, in this paper, we present an alternative algorithm that requires no replication. We experiment with data distributions that can lead to increased replication using the previously proposed algorithms and we show the benefits of avoiding replication in such cases.

4.3 Size Separation Spatial Join

Size Separation Spatial Join derives its properties from the Filter Tree join algorithm [SK96], which was presented in Chapter 3. Filter Trees partition spatial data sets by size. Given two spatial data sets, their join can be computed with minimal effort, requiring that each page of the data sets be read only once. $S^3J$ constructs a Filter Tree partition of the space on the fly without building complete Filter Tree indices. The level $j$ filter is composed of $2^j - 1$ equally spaced lines in each dimension. The level of an entity is the highest one (smallest $j$) at which the MBR of the entity is intersected by any line of the filter. This assures that large entities are caught at high levels of the Filter Tree, while most small entities fall to lower levels.

4.3.1 $S^3J$ Algorithm

Denoting the opposite corners of the MBR of an entity by $(x_l, y_l)$ and $(x_h, y_h)$, $S^3J$ uses two calculated values:

- Hilbert$(x_c, y_c)$, the Hilbert value of the center of the MBR (where $x_c = \frac{x_l + x_h}{2}$, $y_c = \frac{y_l + y_h}{2}$) [Bia69].

- Level$(x_l, y_l, x_h, y_h)$, the level of the Filter Tree at which the entity resides (which is the number of initial bits in which $x_l$ and $x_h$ as well as $y_l$ and $y_h$ agree) [SK96].

Given two spatial data sets, $A$ and $B$, $S^3J$ proceeds as follows. Each data set in turn is scanned and partitioned into level files. For each entity, its level, Level$(x_l, y_l, x_h, y_h)$, is determined, and an entry of the form shown in figure 4.5 is composed and written to the corresponding level file for that data set.
The memory requirement of this phase is just $L + 1$ pages where $L$ is the number of level files (typically, 10 to 20) for the data set being partitioned. One page is used for reading the data set, and $L$ are used for writing the level files.

Next, each level file for each data set is sorted so that the Hilbert values of the entries are monotonically nondecreasing. The final step of the algorithm is to join the two sets of sorted level files. The join is accomplished by performing a synchronized scan over the pages of all level files and reading each page once, as follows: Let $A^l(H_s, H_e)$ denote a page of the $l$-th level file of $A$ containing entities with Hilbert values in the range $(H_s, H_e)$. Then for level files $l = 0, \ldots, L$:

- process entries in $A^l(H_s, H_e)$ with those contained in $B^{l-i}(H_s, H_e)$ for $i = 0, \ldots, l$.
- process entries in $B^l(H_s, H_e)$ with those in $A^{l-i}(H_s, H_e)$ for $i = 1, \ldots, l$. 

Figure 4.4: Space Partition by $S^3 J$

Figure 4.5: Format of entries in a level file
Figure 4.4 shows two levels of the space segmentation on which $S^3J$ is based and presents the intuition behind the algorithm. $S^3J$ divides the space in multiple resolutions as opposed to PBSM and SHJ which partition the object space at a single level. $S^3J$ takes advantage of this space partitioning scheme and is able to perform the join while reading each page only once. Partitioning the space in multiple resolutions and placing each object at a level determined largely by its size, the algorithm can determine which pages are actually needed at each step. Figure 4.4 presents two data sets, $A$ and $B$, each composed of two level files after being processed by $S^3J$. Partition $A_1$ from data set $A$ needs to be processed against partitions $B_1$ and $B_0$ of data set $B$ only. Similarly, partition $B_1$ of data set $B$ has to be processed only with partition $A_0$ of $A$. No further processing for these partitions is necessary since no other overlapping pairs are possible.

Figure 5.4 summarizes the $S^3J$ algorithm. Notice that no assumptions about the statistical properties of the data set are made. The algorithm can be applied either to base spatial data sets or to intermediate data sets without any modification. While we choose to use Hilbert curves to order level files, any curve that recursively subdivides the space will work (e.g., z-order, gray code curve, etc.). Notice that the computation of the Hilbert value is not always necessary. The Hilbert values can be computed at the time entities are inserted into the file and become a part of the descriptors of each spatial entity at the expense of storing them. For base spatial data sets this is probably a good choice. When the spatial data sets involved are derived from base sets via a transformation that changes the entity's physical position in the space or creates new entities, the Hilbert values can be recomputed.

The implementation of the $S^3J$ algorithm is straightforward. Partitioning the data sets involves only reading each entity descriptor and routing it to the appropriate level file (buffer page) based on examining the bit representations of the coordinates of the corners of its MBR. Sorting each level file, based on the Hilbert value of the center of the MBR of each entity, can be done with a sort utility commonly available in database systems. Finally, the synchronized scan of the level files strongly resembles an $L$-way merge sort (which can be implemented in a couple hundred lines of code).
Given two spatial data sets $A$ and $B$,

- Scan data sets $A$ and $B$ and for each entity:
  
  1. Compute the Hilbert value of the entity, $H(x, y)$.
  
  2. Determine the level at which the entity belongs and place its entity descriptor in the corresponding level file.

- For each level file, sort by Hilbert value.

- Perform a synchronized scan over the pages of level files.

Figure 4.6: Size Separation Spatial Join Algorithm

4.3.2 Dynamic Spatial Bitmaps for Filtering

Both PBSM and SHJ are capable of filtering, which makes it possible to reduce the size of the input data sets during the partitioning phase. $S^3J$ as described, performs no filtering since the partitioning of the two data sets is independent. No information obtained during the partitioning of the first data set is used during the partitioning of the second.

$S^3J$ can be extended to perform filtering by using Dynamic Spatial Bitmaps (DSB). DSB is similar to the technique of bitmap join indices in the relational domain [Val87] [OG95] [O'N96]. However, DSB is tailored to a spatial domain.

$S^3J$ dynamically maps entities into a hierarchy of level files. Given a spatial entity, pages from all the level files of the joining data set have to be searched for joining pairs, but, as indicated in the previous section, this is done in a very efficient manner.

DSB constructs a bitmap representation of the entire data space, as if the complete data set were present in one level file. A bitmap is a compressed indication of the contents of a data set. In the relational domain, using a bitmap of $N$ bits to represent a relation of $N$ tuples, we can perform a mapping between tuples and bits. Using this mapping we can obtain useful information during query processing. For example we could, by consulting the bitmap, check whether tuples with certain attributes exist. Now consider
a two dimensional grid. In a similar manner, we can define a mapping between grid cells
and bits of a bitmap. In this case, the bitmap could, for example, record whether any
entity intersects the grid cell or not.

To support filtering in $S^3J$, we use a bitmap corresponding to level $l$. At level file,
$l$, there are $4^l$ partitions of the space, so the bitmap, $M$, will have $4^l$ one-bit entries.
Initially all the bit entries of $M$ are set to zero. Then, during the partitioning phase, for
each spatial entity, $e$, that belongs to level file $l_e$ and has Hilbert value $H_e^{l_e}$:

- If $l \leq l_e$, we transform the Hilbert value, $H_e^{l_e}$, of $e$ into $H_e^l$ (by setting to zero the
  $l - l_e$ least significant bits of $H_e^{l_e}$). We then set $M[H_e^l]$ to one.

- If $l > l_e$ we have to compute the Hilbert values at level file $l$, $H_{e1}^{l}, H_{e2}^{l}, \ldots, H_{en}^{l}$,
  that completely cover $e$ and set $M[H_{ei}^{l}], i = 1, \ldots, n$ to one. The computation of
  $H_{e1}^{l}, H_{e2}^{l}, \ldots, H_{en}^{l}$ can be performed either by determining all the partitions at level
  $l$ that $e$ overlaps and computing their Hilbert values, or by extending $H_e^{l_e}$ with all
  possible $l_e - l$ bit strings.

The operation described above essentially projects all entities onto level file $l$, the
bitmap level. Then, during the partitioning of the second data set $B$, for each spatial
entity $e$, the same operation is performed, but this time:

- If $l \leq l_e$, $e$ is placed into level file $l_e$ only if $M[H_e^l]$ is set to one.

- If $l > l_e$, $e$ is placed into level file $l_e$ only if at least one of the bits $M[H_{e1}^l], M[H_{e2}^l], \ldots, M[H_{en}^l]$ is set to one.

Figure 4.7 illustrates the operation of Dynamic Spatial Bitmaps. Entities, $e_1$ and $e_2$,
existing in level file $L_2$, are projected to the higher level $L_1$ which, for the purposes
of this example, is the level chosen to represent the bitmap. The corresponding bit, of the
bitmap is set to one, indicating that entities exist in that portion of the space. Similarly,
entity $e_3$ from level file $L_0$ is projected to $L_1$. For $e_3$, since it overlaps partitions 0 and 1 of
$L_1$, only those bits should be set to one. We can either calculate the partitions involved
for each entity and set only the corresponding bits or set all the bits corresponding to
the partition that contains $e_3$ in $L_0$. The latter method is faster but less precise.
Consider again the example in figure 4.4. A spatial entity belonging in partition $B_1$ of data set $B$ needs to be stored in a level file for data set $B$ only if a spatial entity of data set $A$ exists in partition $A_1$ and/or $A_0$. Information about whether any spatial entity of data set $A$ exists in any partition of any level file is captured by the bitmap.

The size of the bitmap depends on which level file is chosen as the base onto which to project the data space. For level file $l$, the size of the bitmap is $4^l$ bits. With a page of size $2^p$ bits, $2^{2l-p}$ pages are needed to store the bitmap. Assuming a page size of $2^{12}$ bits (4KB), using level file seven for bitmap construction will yield a bitmap of four pages. Using level eight will yield a bitmap of sixteen pages and so on. There is a tradeoff between the size of the bitmap and its effectiveness. Using a lower level file (larger $j$) will yield a more precise bitmap. However, this will increase the number of pages needed to store the bitmap as well as the processor time to manipulate it. As long as a spatial entity belongs in a level lower than the level file used to represent the bitmap, the Hilbert value transformation is very fast, since it involves a simple truncation of a bit string. However, for spatial entities belonging to level files higher than the bitmap level file, several ranges of Hilbert values have to be computed, requiring more processor time.
Table 4.1: Symbols and their meanings

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_f$</td>
<td>Size of File $f$ in pages</td>
<td>$M$</td>
<td>Memory Size in Pages</td>
</tr>
<tr>
<td>$J$</td>
<td>Size of join result in pages</td>
<td>$r_f$</td>
<td>replication factor for data set $f$</td>
</tr>
<tr>
<td>$D$</td>
<td>Divisions of space</td>
<td>$L_f$</td>
<td>Number of level files for data set $f$</td>
</tr>
<tr>
<td>$H$</td>
<td>Processor time to compute a Hilbert value</td>
<td>$C$</td>
<td>Size of candidate pair list before sort</td>
</tr>
<tr>
<td>$E$</td>
<td>Object descriptor entries per page</td>
<td>$B$</td>
<td>Size of bulk reads from disk</td>
</tr>
</tbody>
</table>

Alternatively, one might choose to extend $H_{l_e}$ with all possible $l - l_e$ long bit strings. This will offer a fast Hilbert value transformation, since only a bit expansion is involved, but will decrease the precision of the bitmap.

Several options for advanced spatial query processing are opened by the use of DSB. For example, if bitmaps exist for both data sets, a bitmap intersection operation will identify the exact ranges of interest to the join. $S^3J$ could take advantage of these ranges while performing a synchronized scan over the pages of the data sets.

### 4.4 Analysis of I/O behavior

In this section we present an analytical comparison of the I/O behaviors of $S^3J$, PBSM and SHJ. Table 4.1 summarizes the symbols used and their meaning. For the purpose of this analytic comparison, we assume a spatial data set composed of entities with square MBRs of size $d \times d$ that are uniformly distributed over the unit square.

#### 4.4.1 Analysis of the three algorithms

$S^3J$ I/O analysis

The Size Separation Spatial Join algorithm proceeds by reading each data set once and partitioning essentially according to size, creating $L_A + L_B$ level files. The number of page reads and writes for data sets $A$ and $B$ in the scan phase will be:

$$2 S_A + 2 S_B$$

(4.1)

The factor of two accounts for reading and writing each data set.
In the sort phase, $S^3J$ sorts each level file. Assuming a uniform distribution of squares, level file $i$ will contain a fraction of objects given by:

$$f_i = \begin{cases} 
  d(2 - d) & i = 0 \\
  2^i d(1 - \frac{3}{4}2^i d) & i = 1, \ldots, k(d) - 1 \\
  (1 - \frac{1}{2}2^d d)^2 & i = k(d)
\end{cases}$$  \hspace{1cm} (4.2)

where $k(d) = \lfloor - \log_2 d \rfloor$ is the lowest level to which any $d \times d$ object can fall (since $d$ must be less than $2^{-k}$) [SK96]. Then the expected size of each level file $i$ for data set $j$ will be about $S_{ij} = f_i S_j$, $i = 1 \ldots \max(L_A, L_B)$, $j \in A, B$. Assuming that read requests take place in bulks of $B$ pages from the disk, applying merge sort on the level file of size $S_{ij}$ will yield a sort fan-in $F$ of $\frac{M}{B}$ and $[l_i = \log_F S_{ij}]$ merge sort levels ($l_i$ will not commonly be one). The total number of page reads and writes of the sorting process is given by:

$$2 \sum_{i=1}^{L_A} l_A S_{iA} + 2 \sum_{i=1}^{L_B} l_B S_{iB}$$  \hspace{1cm} (4.3)

Once the sorted level files are on disk, $S^3J$ proceeds with the join phase by reading each page only once, computing and storing the join result, incurring:

$$S_A + S_B + J$$  \hspace{1cm} (4.4)

page reads and writes. The total number of page reads and writes of $S^3J$ is the sum of the three terms above, thus total number of I/Os for $S^3J$ is:

$$3S_A + 3S_B + 2 \sum_{i=1}^{L_A} l_A S_{iA} + 2 \sum_{i=1}^{L_B} l_B S_{iB} + J$$  \hspace{1cm} (4.5)

The best case for $S^3J$ occurs if each level file fits in main memory (i.e., $S_{ij} \leq M, \forall i$). In this case the total number of page reads and writes of the algorithm becomes:

$$5S_A + 5S_B + J$$  \hspace{1cm} (4.6)

In its worst case, $S^3J$ will find only one level file in each data set. In this case, the total number of page reads and writes will be:

$$3S_A + 3S_B + 2l_A S_A + 2l_B S_B + J$$  \hspace{1cm} (4.7)
Except for artificially constructed data sets, the largest of the level files would usually contain 10% to 30% of the entities in the data sets. If the Hilbert values are initially not part of the spatial entity's descriptor, then they have to be computed. This computation takes place while partitioning the data sets into levels. The processor time for this operation is:

\[ H(S_A + S_B)E \]  

(4.8)

Using a table driven routine for computing the Hilbert values, we were able to perform the computation in less than 10 \( \mu \)sec per value at maximum precision on a 133MHz IBM RS6000 processor, so \( H \approx 10 \mu \)secs.

**PBSM I/O analysis**

The number of partitions suggested by Patel and DeWitt for the PBSM algorithm [PD96] is:

\[ D = \frac{S_A + S_B}{M} \]  

(4.9)

Defining the replication factor \( r_f \) as:

\[ r_f = \frac{\text{Data set size after replication and filtering}}{\text{Original data set size (S)}} \]  

(4.10)

the number of page reads and writes during the partitioning phase is:

\[ (1 + r_A) S_A + (1 + r_B) S_B \]  

(4.11)

since the algorithm reads each data set and possibly introduces replication for entities crossing partition boundaries.

Entity replication will increase the data set size, making \( r_f \) greater than one, but filtering, will counteract that, reducing \( r_f \), possibly to be even less than one for cases where the join is highly selective (i.e., where there are very few join pairs). Due to replication, the size of the output file that is written back to disk may be larger than the initial data set size. More precisely, if \( A \) is the data set that is partitioned first, then \( r_A \geq 1 \) and \( r_B \geq 0 \). The amount of replication introduced depends on the data
distributions of the data sets and the degree of dividing of the data space into tiles. Depending on data distributions, \( 1 \leq r_A \leq D \) and \( 0 \leq r_B \leq D \). Notice that \( r_B \) could be less than one depending on the partitioning imposed on the first data set. To illustrate the effects of replication, again assume uniformly distributed squares of size \( d \times d \), normalized in the unit square. Then assuming a regular partitioning of the unit square into sub-squares of side \( 2^{-j} \), the fraction, \( N \), of objects falling inside tiles will be:

\[
1 - d^{2j+1} + d^2 2^{2j}
\]  

(4.12)

assuming that \( d \leq 2^{-j} \), so that the side of each square object is less than or equal to the side of each tile. As a result the fraction of objects replicated will be \( d^{2j+1} - d^2 2^{2j} \). The amount of replication taking place depends on \( d^2 j \), since replication is introduced either by increasing the object size for constant number of tiles or by increasing the number of tiles for constant object size. Figure 4.8 shows the fraction of objects replicated as a function of \( d^2 j \). As \( d^2 j \) increases, the amount of replication that takes place increases.

The algorithm then checks whether corresponding partitions fit in main memory. Assuming that partitions have the same size and that each pair of partitions fits in main memory, the number of page reads and writes for this step is:

\[
r_A S_A + r_B S_B + C
\]

(4.13)

where \( C \) is the size of the initial candidate list. If partition pair \( i \) does not fit in
main memory, then it has to be repartitioned. Using equation (4.9) to compute the number of partitions we expect under a uniform distribution, half the partitions to require repartitioning. Using a hash function to map tiles to partitions, we expect the MBRs of partitions to be the same as the MBR of the original data file. Thus, the fraction of replicated objects remains the same for subsequent repartitions. The total number of page I/Os during the first partitioning phase is given by equation (4.11). Since on average half of the partitions will have to be repartitioned, the expected number of page I/Os during the second partitioning phase will be:

$$\frac{(1 + r_A)r_A S_A}{2} + \frac{(1 + r_B)r_B S_B}{2}$$  \hspace{1cm} (4.14)

For uniform data distributions, this is expected to offer acceptable size balance across partitions and pairs of corresponding partitions will fit in main memory. The algorithm proceeds to read all pairs of corresponding partitions and join them in main memory using plane sweep. The total number of page I/Os for this phase will be:

$$\frac{(1 + r_A)r_A S_A}{2} + \frac{(1 + r_B)r_B S_B}{2} + C$$  \hspace{1cm} (4.15)

where $C$ is the size of the candidate list. After the join phase, the result of the join is stored on disk, but duplicate elimination must be performed since replication of entities may have occurred in both data sets. Duplicate elimination is achieved by sorting the join result. The number of page reads and writes during the sort is:

$$2 J \sum_{i=0}^{l-1} \frac{C}{M} = 2 J \frac{C}{M} \frac{(1 - \frac{1}{F})}{1 - \frac{1}{F}}$$  \hspace{1cm} (4.16)

where $F$ is the fanout factor of the sort. The number of sort merge phases will be $l = \log_F C$. Since elimination of duplicates can take place in any phase of the sort, we have to perform the summation over all sort merge phases, resulting in equation (4.16). If $C$ fits in memory, the cost of page reads and writes during the sort (with duplicate elimination) will be $C + J$.

The total number of page reads and writes of the algorithm results if we sum all components above, taking into account whether intermediate results fit in main memory or not. The replication factors, $r_A$ and $r_B$, play an important role in the total number
of I/O's given above. Their value depends on the number of tiles in the space and the input data distributions. Thus the total number of I/Os for PBSM in the general case (pairs of partitions do not fit in main memory, thus repartitioning takes place) is:

\[ 2(1 + r_A) + 2(1 + r_B) + 2 J \frac{C}{M} \frac{1 - \frac{1}{F}}{1 - \frac{1}{F}} \]  \hspace{1cm} (4.17)

**Spatial Hash Joins**

Assuming that data set \( A \) is to be processed with \( D \) partitions, the number of page reads and writes during sampling and partitioning of data set \( A \) is:

\[ c \ D + 2 \ S_A \]  \hspace{1cm} (4.18)

where \( c \) is some integer and \( cD \) represents (an upper limit on) the random I/O performed while sampling set \( A \). The number of page reads and writes during partitioning of data set \( B \) is:

\[ (1 + r_B) \ S_B \]  \hspace{1cm} (4.19)

since all of data set \( B \) must be read and multiple \( r_B \) of its initial size must be written. After the partitioning phase, the algorithm joins the corresponding pairs of partitions. If the corresponding partitions for both data sets fit in main memory, both partitions will be read and then joined. The join can be done either using nested loops or by constructing an R-tree in main memory for the first partition and probing it with the elements of the second. If both partitions fit in main memory the number of page reads and writes during the join phase is:

\[ S_A + r_B \ S_B + J \]  \hspace{1cm} (4.20)

where the first two terms correspond to reads and the third to writes. However, with SHJ, there is no guarantee that the partitions will be balanced in size or that they will fit in main memory. Moreover, the partition placement depends only on samples taken from one data set. A general analysis of SHJ is difficult, because its behavior depends on the distributions of the joined data. For uniformly distributed squares, an analysis similar to the one presented for PBSM can be applied. However, for specific data set
sizes and main memory size, the number of partitions used by SHJ is much larger than
the number used for PBSM. Consequently, the amount of replication required in SHJ is
expected to be larger than that in PBSM. Assuming that partitions do not fit in main
memory and that partitions are joined using nested loops, the number of page reads and
writes during the join phase becomes:

\[ \sum_{i=1}^{D} \left( \frac{S_{iA}}{M} S_{iB} + S_{iA} \right) \]  \hspace{1cm} (4.21)

where \( S_{iA}, S_{iB} \) are the sizes of the partitions for A and B. Very little can be said about
\( S_{iA} \) and \( S_{iB} \). For uniformly distributed data sets, we expect \( S_{iA} = \frac{S_A}{D} \) and \( S_{iB} = r_B \times \frac{S_B}{D} \).
Thus the total number of I/Os for SHJ is:

\[ cD + 2S_A + (1 + r_B)S_B + \sum_{i=1}^{D} \left( \frac{S_{iA}}{M} S_{iB} + S_{iA} \right) \]  \hspace{1cm} (4.22)

For SHJ, replication is introduced only for one of the two data sets involved. As in
the case of PBSM, the value for the replication factor \( r_B \) plays an important role in the
algorithm’s performance. Notice that in the worst case \( r_B = D \).

4.4.2 Analytical Comparison of the Algorithms

Using the formulas derived in the previous subsections, we perform an analytical com-
parison of the algorithms in terms of I/O behavior. The results are shown in figure 4.9.
For this comparison, we assume that \( S_A = S_B = 10000 \) pages. For \( S^3J \), an adver-
sarial placement of objects based on knowing the position of the underlying grid could lead to
a single level file with all objects in it. However if the adversary does not have knowledge
of the exact position of grid lines (i.e., the grid can be shifted slightly), then even by
placing objects in a non-independent fashion, extreme imbalance of level file size cannot
be forced. Consequently, we take as a practical “worst case” a situation where 30% of the
entities are in the largest of the level files. (This choice is supported by previous analysis
[SK96].) To simplify the figures we assume that 20 duplicate join tuples are introduced
during PBSM’s operation, so the size of the intermediate result is equal to the size of the
join result (that is, \( C = J \)). This favors PBSM, since fewer I/O operations are necessary,
both when writing the intermediate result, and when sorting to eliminate duplicates.
Figure 4.9 presents the total number of page reads and writes predicted by the formulas, normalized to $S^3J$ performance, for PBSM and SHJ as a function of the replication factors, $r_A$ and $r_B$. This is a low selectivity join, since the size of the join result is twice the aggregate size of the two data sets. A word of caution is in order when interpreting figure 4.9: For two specific data sets, there is no reason to believe that the replication factor for the second data set, $r_B$, introduced by PBSM and SHJ is the same. As we pointed out earlier, we expect replication to be higher for SHJ. However, in order to present a comparison, we represent $r_B$ on a common axis for both algorithms.

$S^3J$ does not introduce any replication so its performance is independent of replication factors. In addition, the buffering requirements of the algorithm during the partitioning and join phases are modest, being only a page per level file plus a page in the buffer pool for output tuples: $L_A + L_B + 1$. Increasing the available buffer size helps the sorting phase of $S^3J$ (if the level files cannot fit in main memory).

In contrast, PBSM's performance depends heavily on the degree of replication as is evident in figure 4.9. As replication increases the performance of PBSM becomes worse. PBSM makes the partitions fit in main memory only when partitions are size balanced and the aggregate size of the data sets before partitioning is less than or equal to their aggregate size after partitioning. This is due to the fact that the number of partitions used for PBSM (equation (4.9)) does not take replication into account. It sets the number of partitions based on the size of the data sets before partitioning. When we increase the amount of memory available to the join, the part of PBSM's total number of page reads and writes that is affected, is the number of I/Os incurred while sorting. Since the sort can use a larger fanout, the number of I/Os is smaller.

SHJ introduces replication only in the second data set and, as a result, its I/O behavior is competitive to $S^3J$ for higher replication factor values, as shown in figure 4.9. For the purposes of this analysis, each random disk access incurred via sampling during the partitioning phase of SHJ is assumed five times more expensive than the cost per page in a sequential access. SHJ typically uses a large number of partitions and thus frequently makes pairs of partitions fit in main memory during the join phase. Increasing the buffer size available by a factor $f$, causes an increase in the number of partitions by a factor
which can be more than \( f \) timer the estimated before the increase. This will improve the join phase of the algorithm but will slow down its partitioning phase.

### 4.5 Experimental Comparison

In this section, we present experimental results from prototype implementations of all three algorithms. We include experimental results based on both of real and synthetic data sets. We implemented all three algorithms on top of a common storage manager that provides efficient I/O. Several components common to all algorithms were shared between implementations, contributing to the fairness of the comparison of the algorithms at the implementation level. Specifically, the same sorting module is used by \( S^3J \) and PBSM, and all three algorithms use the same module for plane sweep. All algorithms were implemented in the C programming language, under the AIX operating system. The size of our \( S^3J \) implementation is approximately 600 lines of C code.

All of our experiments were conducted on an IBM RS6000 model 43P (133MHz), running AIX with 64MB of main memory (varying the buffer size during experiments)
with a Seagate Hawk 4 disk with capacity 1GB attached to it. The processor’s SPEC ratings are SPECint95 4.72 and SPECfp95 3.76. Average disk access time (including latency) is 18.1 msec assuming random reads.

We present and discuss sets of experiments, treating joins of both synthetic and real data sets for low (many output tuples) and high (few output tuples) selectivity joins. For our treatment of $S^3J$, we assume that the Hilbert value is computed dynamically. If the Hilbert value were present in the entity descriptor initially, the response times for $S^3J$ would be smaller than the ones presented here by a small amount, reflecting savings of processor time to compute the values.

For PBSM, we demonstrate the effect of different parameters on the performance of the algorithm. We include results for various numbers of tiles. In all PBSM experiments, we compute the number of partitions using equation (4.9) as suggested by Patel and DeWitt [PD96]. Similarly, SHJ performance depends on the statistical properties of the input data sets. We compute the number of partitions using the formula suggested by Lo and Ravishankar [LR95].

We present the times required for different phases of the algorithms. Table 4.2 summarizes the work done in each phase phase for the three algorithms. For the experiments that follow, unless stated otherwise, the total buffer space available is 10% of the total size of the spatial data sets being joined.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Phase</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^3J$</td>
<td>Partition</td>
<td>Reading, partitioning and writing the level files for both data sets</td>
</tr>
<tr>
<td></td>
<td>Sort</td>
<td>Sorting (reading and writing) the sorted level files</td>
</tr>
<tr>
<td></td>
<td>Join</td>
<td>Merging the sorted level files and writing the result on disk</td>
</tr>
<tr>
<td>PBSM</td>
<td>Partition</td>
<td>Reading, partitioning and writing partitions for both data sets</td>
</tr>
<tr>
<td></td>
<td>Join</td>
<td>Joining corresponding partitions and writing the result on disk</td>
</tr>
<tr>
<td></td>
<td>Sort</td>
<td>Sorting the join result with duplicate elimination and writing the result on disk</td>
</tr>
<tr>
<td>SHJ</td>
<td>Partition</td>
<td>Reading, partitioning and writing partitions for both data sets</td>
</tr>
<tr>
<td></td>
<td>Join</td>
<td>Joining corresponding partitions and writing the result on disk</td>
</tr>
<tr>
<td></td>
<td>Sort</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 4.2: Phase Timings for the three algorithms
Table 4.3: Real and Synthetic Data Sets used

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>UN1</td>
<td>Uniformly-Distributed Squares</td>
<td>100,000</td>
<td>0.4</td>
</tr>
<tr>
<td>UN2</td>
<td>Uniformly-Distributed Squares</td>
<td>100,000</td>
<td>0.9</td>
</tr>
<tr>
<td>UN3</td>
<td>Uniformly-Distributed Squares</td>
<td>100,000</td>
<td>1.6</td>
</tr>
<tr>
<td>LB</td>
<td>Line Segments from Long Beach County, California</td>
<td>53,145</td>
<td>0.15</td>
</tr>
<tr>
<td>MG</td>
<td>Line Segments from Montgomery County, Maryland</td>
<td>39,000</td>
<td>0.12</td>
</tr>
<tr>
<td>TR</td>
<td>Squares of Various Sizes</td>
<td>50,000</td>
<td>13.96</td>
</tr>
<tr>
<td>CFD</td>
<td>Point Data (CDF)</td>
<td>208,688</td>
<td>-</td>
</tr>
</tbody>
</table>

4.5.1 Description of Data Sets

Table 4.3 describes the data sets used for our experiments. All the data sets composed of uniformly distributed squares are normalized in the unit square. UN1, UN2 and UN3 have artificially low variability of the sizes of objects and consequently low coverage, 0.4, 0.9 and 1.6 respectively. The LB and MG data sets contain road segments extracted from the TIGER/Line data set [Bur91]. The first (LB) contains lines corresponding to road segments in Long Beach County, California. The second (MG) contains similar data for road segments from Montgomery County, Maryland. Data set TR is used to model scenarios in which the spatial entities in the data sets are of various sizes. We produced a data set in which the sizes of the square spatial entities are generated according to a triangular shaped distribution. More precisely, the size of the square entities is \( d = 2^{-l} \) where \( l \) is drawn from a probability distribution with minimum value \( x_1 \), maximum value \( x_3 \), and the peak of the triangular distribution at \( x_2 \), which is shown in figure 4.10. As one would expect, the overlap among the entities of such a data set is high. TR contains 50,000 entities and was generated using parameters \( x_1 = 4, x_2 = 18, x_3 = 19 \). CFD is a vertex data set from a Computational Fluid Dynamics model, in which a system of equations is used to model the air flows over and around aero-space vehicles. The data set describes a two dimensional cross section of a Boeing 737 wing with flaps out in landing
configuration. The data space consists of a collection of points (nodes) that are dense in areas of great change in the solution of the CFD equations and sparse in areas of little change. The location of the points in the data set is highly skewed.

### 4.5.2 Experimental Results

**No Filtering Case**

We present and discuss a series of experiments involving low selectivity joins of synthetic and real data sets. Table 5.3 summarizes all the experimental results in this subsection and presents the response times of PBSM and SHJ normalized to the response time of $S^3J$ as well as the replication factors observed in each case.

The first two experiments involve data objects of a single size that are uniformly distributed over the unit square. For uniformly and independently distributed data, the coverage of the space is a realistic measure of the degree of overlap among the entities of a data set. From the first experiment to the second, we increase the coverage (by using squares of larger size) of the synthetic data sets and present the measured performance of the three algorithms. For algorithms that partition the space and replicate entities across partitions, the probability of replication increases with coverage, for a fixed number of partitions.

Figure 4.11a presents the response time for the join of two uniformly distributed data sets, UN1 and UN2 containing 100,000 entities each. Results for PBSM are included
(a) UN1 join UN2 (coverage 0.4 and 0.9).

(b) UN2 join UN3 (coverage 0.9 and 1.6).

Figure 4.11: Join performance for uniformly distributed data sets of squares
Table 4.4: Join response times, normalized to $S^3J$ observed response time and replication. (NRT: Normalized Response Time)

for two different choices of tiling: the first choice is the number of tiles that achieves satisfactory load balance across partitions and the second is a number of tiles larger than the previous one. For $S^3J$ the processor time needed to evaluate the Hilbert values accounts for 8% of the total response time. The partitioning phase is relatively fast, since it involves sequential reads and writes of both data sets while determining the output level of each spatial entity and computing its Hilbert value.

For PBSM, since we are dealing with uniformly distributed objects, a small number of tiles is enough to achieve balanced partitions. The greatest portion of time is spent partitioning the data sets. Most partition pairs do not fit in main memory and the algorithm has to re-read and repartition those that do not fit in main memory. Approximately half of PBSM’s response time is spent partitioning the input data sets and the rest is spent joining the data sets and sorting (with duplicate elimination) the final output.

SHJ uses more partitions than PBSM does for this experiment. The large number of partitions covers the entire space and introduces overlap between partition boundaries. The algorithm spends most of its time sampling and partitioning both data sets. As is evident from figure 4.11a, the partitioning phase of SHJ is more expensive than the corresponding phase of $S^3J$, and a little more expensive than that of PBSM with large
tiles. The join phase, however, is fast since all pairs of partitions fit in main memory, and, due to less replication, fewer entities have to be tested for intersection.

Figure 4.11b presents the results for the join of UN2 and UN3. The impact of higher coverage in UN3 relative to UN1 affects $S^3J$ only in processor time during the join phase. The portion of time spent partitioning into levels and sorting the level files is the same. Although the partitioning times remain about the same, join time and sorting time increase according to the data set sizes. For SHJ the larger replication factor observed increases I/O as well as processor time in the partitioning and join phases. Due to the increased replication, the join phase of SHJ is more costly than in the previous experiment.

Figures 5.15a and 5.15b present results for joins of data sets LB and MG. For each of LB and MG, we produce a shifted version of the data set, LB' and MG', as follows: the center of each spatial entity in the original data set is taken as the position of the lower left corner of an entity of the same size in the new data set.

Figure 5.15a presents performance results for the join of LB and LB'. For $S^3J$, the time to partition and join is a little more than the time to sort the level files. When decomposed by $S^3J$, LB yields 19 levels files. The largest portion of the execution time is spent joining partition pairs. PBSM’s performance is worse with more tiles due to increased replication. In this case, the join result is larger than both input data sets, so PBSM incurs a larger number of I/Os from writing the intermediate result on disk and sorting it. Not all partitions fit in main memory (because of the non-uniformity of the data set) and SHJ has to read pages from disk during the join phase. Figure 5.15b presents the corresponding experiment involving the MG and MG' data sets. Similar observations hold in this case.

The experiments described above offer intuition about the trends and tradeoffs involved with real and synthetic data sets with moderate and low coverage. With the following experiment, we explore the performance of the algorithms on data sets with high coverage, with varying sizes in the spatial entities, and with distributions with high clustering.

Figure 4.13a presents the results of a self join of TR. Although only a single data set is
Figure 4.12: Join performance for real data sets

(a) LB and LB' join.

(b) MG and MG' join.
(a) Triangular distribution, self join.  

(b) CFD data set self join.  

Figure 4.13: Self Join performance for real data sets
involved, the algorithm does not exploit that fact. \( S^3J \), with Hilbert value computation, is processor bound. Due to the high coverage in the data set, \( S^3J \) has to keep the pages of level files in memory longer while testing for intersections.

PBSM spends most of its time partitioning and joining corresponding partitions, but sorting and duplicate elimination also account for a large fraction of the execution time, since the size of the join result is large. In contrast with \( S^3J \), PBSM appears I/O bound.

SHJ requires extensive replication during the partitioning of the second data set. This results from the spatial characteristics of the data set and the large number of partitions used. Large variability in the sizes of the entities leads to large partitions. As a result, the probability that an entity will overlap more than one partition increases with the variability of the sizes of the spatial entities. SHJ is I/O bound on our experimental setting and most of its time is spent joining pairs of partitions which, in this case, do not fit in main memory. Due to the replication, the time spent by the algorithm partitioning the second data set is much larger than the time spent during the partitioning of the first data set. Although SHJ introduces more replication than PBSM, it does not require duplicate elimination, and, depending on the amount of replication and repartitioning performed by PBSM, its partitioning phase might be cheaper. It is due to the fact that no duplicate elimination is needed that SHJ is able to outperform PBSM in the case of large tiles.

Figure 4.13b presents results from a self join of CFD. We employ a spatial join to find all pairs of points within distance \( 10^{-6} \) from each other. For this data distribution, which involves a large cluster in the center of the data space, both PBSM and SHJ perform poorly. PBSM requires a large number of tiles to achieve load balancing for its partitions, and a lot of repartitioning takes place, introducing a large degree of replication. The join phase is faster than that for, SHJ, however since all pairs of partitions obtained via repartitioning fit in main memory in this experiment. The sampling performed by SHJ is ineffective in this case, and the join phase is costly involving a large number of page reads from the disk. The partitions have varying sizes, and one of them contains almost the entire data set.
The Effects of Filtering

With the experiments described in the previous subsection, we investigated the relative performance of the algorithms when no filtering takes place during the join of the data sets involved. With the experiments in this section, we study the effects of filtering on all three algorithms.

For this purpose, we perform two experiments. In the first, we join two uniformly distributed data sets of 50,000 entities each. The data sets are uniformly distributed in disjoint portions of the unit square, so that there is no overlap between the total MBRs of the two data sets. The results are presented in figure 4.14a.

$S^3J$ incurs a number of disk accesses while reading and partitioning the first data set, and requires processor time to create the bitmap for it. During processing of the second data set, the algorithm has to read the data set, but no entities are written back on disk, since the complete data set is filtered out with the use of the bitmap constructed during the partitioning of the first data set. PBSM reads the first data set and introduces a 4% replication factor ($r_A = 1.04$) during its partitioning. The second data set is completely filtered out for PBSM as well. PBSM incurs a larger number of I/Os than $S^3J$. SHJ has to go through its partitioning phase for the first data set. This involves sampling the first data set and populating its partitions. The second data set is filtered out completely by SHJ as well. However, the expensive partitioning phase causes the performance of SHJ to be worse relative to PBSM in this case.

Figure 4.14b presents the results of a second experiment involving uniformly distributed squares, but this time, the total MBR of the first data set is contained in the total MBR of the second. The ratio of the area of the total MBR of the two data sets, is one thousand. There exist some joining entities between the two data sets. $S^3J$ reduces the size of the second data set using the bitmap, and the total number of page reads and writes in its partitioning, sorting and joining phase is small. PBSM is able to nearly match $S^3J$'s performance, since a small replication factor is introduced during the partitioning of the first data set, and a large portion of the second data set is successfully filtered out and not involved in further processing. For SHJ, although a significant
Figure 4.14: Join performance with filtering for uniformly distributed data sets of squares.
amount of filtering takes place, the expensive partitioning phase once again causes overall poor performance.

4.5.3 Discussion

We have presented several experiments comparing the performance of the three algorithms, $S^3J$, PBSM, and SHJ, involving real and synthetic data sets. Our experimental results are consistent with our analytic observations. The relative performance of the algorithms depends heavily on the statistical characteristics of the datasets. $S^3J$ appears to have comparable performance to SHJ when the replication introduced is not large, but is able to outperform it by large factors as replication increases. PBSM is comparable with $S^3J$ when sufficient filtering takes place and, in this case, PBSM performs better than SHJ. The amount of filtering that makes PBSM competitive is difficult to quantify, because it depends on the characteristics of the data sets involved, the amount of replication that PBSM introduces, the order in which the data sets are partitioned, and the number of page reads and writes of the sorting phase of PBSM.

While $S^3J$ neither requires nor uses statistical knowledge of the data sets, the best choice for the number of tiles in PBSM or for the amount of sampling in SHJ depends on the spatial characteristics of the data sets involved in the join operation. Good choices can be made only when statistical information about the data sets is available and the MBRs of the spaces are known. When entities are located independently under uniform distributions, the amount of overlap between the MBRs of the two spaces gives a good estimate of the expected size of the join result. Under skewed data distributions however, no reliable estimate can be made, unless detailed statistical characteristics of both data sets are available. We believe that such measures could be computed for base spatial data sets. However, for intermediate results, the number of page reads required to estimate the statistical characteristics might be high.

It appears from our experiments that, although the partitioning phase of SHJ is expensive, it is worthwhile in the case of low selectivity joins, because it yields a large number of partitions which usually fit in main memory in the subsequent join phase. In contrast, the analytical estimate for the number of partitions of PBSM doesn't con-
consistently yield appropriate values. The partition pairs often do not fit in main memory because of the replication introduced by the algorithm, and the cost of repartitioning can be high.

We experimentally showed that there are data distributions (such as the triangular data distribution we experimented with) for which both PBSM and SHJ are very inefficient. For such distributions it is possible that, due to the high replication introduced by both PBSM and SHJ, the disk space used for storing the replicated partitions might be exhausted. This is also possible for the output of the join before the duplicate elimination in the case of PBSM, especially in environments with limited disk space.

Depending on the statistical characteristics of the data sets involved, $S^3 J$ can be either I/O bound or processor bound. We experimentally showed that, even with distributions with many joining pairs, both PBSM and SHJ are I/O bound, but $S^3 J$ can complete the join with a minimal number of I/Os, and outperforms both other algorithms. For distributions in which filtering takes place, we experimentally showed that $S^3 J$ with DSB is able to outperform both PBSM and SHJ. When enough filtering takes place, PBSM does better than SHJ mainly due to the expensive partitioning phase of SHJ. However, the previous argument depends also on the number of tiles used by PBSM, since it might be the case that excessive replication is introduced by PBSM using too many tiles, and the performance advantages are lost. $S^3 J$ is equally capable of reducing the size of the data sets involved and is able to perform better than both PBSM and SHJ.

4.6 Conclusions about $S^3 J$

We have presented a new algorithm to perform the join of spatial data sets when indices are not available for them. Size Separation Spatial Join imposes a dynamic hierarchical decomposition of the space and permits an efficient joining phase. Moreover, our algorithm reuses software modules and techniques commonly present in any relational system, thus reducing the amount of software development needed to incorporate it. The Dynamic Spatial Bitmap feature of $S^3 J$ can be implemented using bitmap indexing techniques already available in most relational systems. Our approach shows that often the efficient bitmap query processing algorithms already introduced for relational data can
be equally well applied to spatial data types using our algorithm.

We have presented an analytical and experimental comparison of $S^3J$ with two previously proposed algorithms for computing spatial joins when indices are not available for the data sets involved. Using a combination of analytical techniques and experimentation with real and synthetic data sets, we showed that $S^3J$ outperforms current alternative methods for a variety of types of spatial data sets.
Chapter 5

High Dimensional Similarity Joins

5.1 Introduction

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

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

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

Agrawal et al. [AFS93] proposed the use of Fourier transforms to map time series data into points in a multidimensional space. The resulting points are inserted into a multidimensional indexing structure. Given a distance predicate $\epsilon$ and a query point, all the points within distance $\epsilon$ from the query point according to a distance metric can be found by consulting the index. That way, assuming that the query point corresponds to a specific time series and is obtained via the transform, all the time series that are similar to this one will be reported. Faloutsos et al. extend time series similarity to apply to fragments (sub-sequences) of a time series [FRM94]. Each data sequence is fragmented and sequence fragments are mapped into points in a multidimensional space. That way, subsequence similarity queries can be answered. Agrawal et al. extend previous work on time series similarity to make it robust with respect to the presence of noise and scaling [ALSS95]. Rafiei and Mendelzon [RM97] discuss similarity queries in the context of several transformations that allow more elaborate comparisons. Queries on multimedia data bases in general are discussed by Faloutsos [Fal96].

5.2 Problem Statement

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

$$D'_d = \left( \sum_{i=1}^{d} |(x_i - y_i)^p| \right)^{1/p} \leq \epsilon$$

Then $D'_d$ is referred to as Manhattan Distance for $p = 1$, and Euclidean Distance for $p = 2$.

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

In the one dimensional (relational) domain, given two relations, \( A \) and \( B \), the join operation \( A \Join B \) on attributes \( a \) and \( b \) applies predicate \( \theta \) to the attribute values of pairs of tuples, and reports a combined tuple whenever the predicate is true. As was indicated in section 2.1 this is a quadratic operation in a degenerate case.

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

### 5.3 Survey of Various Algorithmic Approaches

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

Algorithms for the multidimensional join problem can be separated into two categories. The first category includes algorithms that treat data sets for which no indices exist. The second category includes algorithms that utilize preconstructed indices to solve the multidimensional join problem.

We describe and analyze four algorithmic approaches from the first category that can provide a solution to the multidimensional join problem: *Brute Force, Divide and Conquer, Replication, and Space Filling Curves*. All four algorithmic approaches use the
following technique to identify points of interest within distance \( \frac{\varepsilon}{2} \) of a given point \( x \). Each point \( x \) can be viewed as the center of a hypercube of side \( \varepsilon \). We refer to the hypercube as the approximation of the multidimensional point. The distance of \( x \) from all the points in the hypercube is computed (based on some distance metric) and only the points within distance \( \varepsilon \) of one another are reported. The volume of the hypercube approximation of a multidimensional point, however, is larger than the volume of the hypersphere centered at the multidimensional point having radius \( \frac{\varepsilon}{2} \). Only points within the hypersphere belong to the join result. Thus, whenever we approximate a multidimensional point with a hypercube an error is introduced. This error causes points in the space belonging in the hypercube but not in the hypersphere, to be reported as within distance \( \frac{\varepsilon}{2} \) from the point being approximated. Let \( Err \) denote the error of the approximation of a hypersphere by a hypercube, defined as:

\[
Err = Volume(Hypercube) - Volume(Hypersphere)
\] (5.2)

When we approximate a Hypersphere of radius \( \frac{\varepsilon}{2} \) with a hypercube of side \( \varepsilon \) in \( d \) dimensions, \( Err \) increases with \( d \) for constant \( \varepsilon \) and increases with \( \varepsilon \) for constant \( d \). Figure 5.1 illustrates the situation in two dimensions. \( Err \) corresponds to the area inside the square, but outside the circle. For the rest of this paper, we assume Euclidean distances, but any other distance metric can be applied instead, without affecting the operation of the algorithms.

Although the problem of searching and indexing in more than one dimension has been studied extensively, no indexing structure is known that retains its indexing efficiency as dimensionality increases. A wide range of indexing structures have been proposed for the two dimensional indexing problem [Sam90]. Although conceptually most of these structures generalize to multiple dimensions, in practice their indexing efficiency degenerates rapidly as dimensionality increases. A recent experimental study by Berchtold et al. [BKK96] showed that using the X-tree, a multidimensional indexing structure based on R-trees [Gut84], several multidimensional queries degenerate to linear search as dimensionality increases. The X-tree [BKK96] was shown experimentally to outperform previously proposed multidimensional structures, like the TV-tree [LJF94].
Figure 5.1: Approximation of a hypersphere in two dimensions

Figure 5.2: R* tree construction cost as dimensionality increases

An algorithm was proposed by Brinkhoff et al. [BKS93] for the two dimensional spatial join problem using R*-trees [BKSS90]. Since the R-tree family is a popular family of indexing structures [Gut84][BKSS90][SRF87], we extended the algorithm of Brinkhoff et al. to multiple dimensions, and we report on its performance in subsequent sections. We believe that the trends in performance we report for the join of multidimensional R*-trees are representative of the join performance of other structures based on the R-tree concepts.

Figure 5.2 presents the time to construct an R* -tree index for 100,000 multidimensional points for various dimensionalities, inserting one point at a time. The construction
time is measured on a 133MHz IBM RS6000 processor (including the time to write filled pages to disk). The cost increases approximately linearly as dimensionality increases, since the work the algorithm performs per point increases as more dimensions are added. Notice that bulk loading of the index requires application of a multidimensional clustering technique, which has high cost as well. Figure 5.2 suggests that, for an on-line solution to the multidimensional join problem, building indices on the fly for non-indexed data sets and using algorithms from the second category to perform the join might not be a viable solution for high dimensionalities due to the prohibitive index construction times.

**5.3.1 Algorithms That Do Not Use Indices**

**5.3.1.1 Brute Force Approach**

**Main Memory Case:** If data sets are small enough to fit in main memory together, both can be read into memory and the distance predicate can be evaluated on all pairs of data elements. Assuming $A$ and $B$ are two multidimensional data sets containing $n_A$ and $n_B$ points respectively, the total cost of this process will be $n_A \times n_B$ predicate evaluations. The cost of each predicate evaluation increases linearly with the dimensionality of the data points. A faster algorithm for the predicate evaluation step is to use a generalization of the *Plane Sweep* technique in multiple dimensions [PS85]. This makes it possible to reduce the number of distance computations by evaluating the predicate only between pairs of multidimensional points for which the corresponding hypercubes intersect. The complexity of a $d$ dimensional sweep involving $O(n)$ points, to report $k$ pairs of overlapping objects is $O(n \log^{d-1} n + k)$ [Mel91]. Note that if two hypercubes of side $2\delta = \epsilon$ overlap, the points at their centers are not necessarily within distance $\epsilon$ of each other. Although the algorithm works well on average, in the worst case all the pairs of distance computations have to be evaluated at a total cost of $n_A \times n_B$ predicate evaluations plus the overhead of the multidimensional sweep.

**Nested Loops (NL):** When both data sets cannot fit in main memory, nested loops is the simplest algorithm to apply [Ull89]. Assuming a buffer space of $M$ pages, the total
I/O cost in page accesses of the join using nested loops will be approximately:

\[ |A| + \frac{|A|}{M - 1}|B| \]  

(5.3)

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

### 5.3.1.2 Divide and Conquer

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

Multidimensional Divide and Conquer Approach (MDC): Multidimensional Divide and Conquer (MDC), is an algorithmic paradigm introduced by Bentley [Ben80], that can be directly applied to the problem at hand. To solve a problem in a multidimensional space, the underlying idea behind the MDC paradigm is to recursively divide the space a dimension at a time, and solve the problem in each resulting subspace. Once the problem is solved in all sibling subspaces, then the solutions of the subspaces are combined in a way specific to the problem under consideration.

Consider the one dimensional case \((d=1)\). Given two sets of \(n\) points on a line, we are to report all pairs of points, one from each set, within distance \(\epsilon\) from each other. We can do this by sorting both sets (an \(O(n \log n)\) operation), and performing a scan of both files by treating portions of each file corresponding to a range of values of the attribute of width \(2\epsilon\). As illustrated in figure 5.3a, both data sets are sorted on increasing value of
Figure 5.3: MDC algorithm for (a) one and (b) two dimensions

the coordinate. By keeping in memory all elements with values in the range 0 to \( \epsilon \) or \( \epsilon \) to \( 2\epsilon \) from both files, we have all the points necessary to correctly report the joining points in the 0 to \( \epsilon \) range that are part of some joining pair. No more points are necessary since any point that joins with a point in the range 0 to \( \epsilon \) must be within distance \( 2\epsilon \) from the left side of the 0 to \( \epsilon \) range. Once we are done with the 0 to \( \epsilon \) range, we can discard the corresponding partitions from the buffer pool and read the next range, \( 2\epsilon \) to \( 3\epsilon \), to finish the processing of the \( \epsilon \) to \( 2\epsilon \) range, and so on. Corresponding ranges in both files can be processed via the plane sweep algorithm.

Letting \( X^2 \) be the pages of \( X \) that contain at least one value in the range \((r_1, r_2)\), the algorithm is shown in figure 5.4. The cost of sorting \( n \) points is \( O(n \log n) \), and, assuming that the cost of scanning and joining both files is \( O(n) \), the algorithm in figure 5.4 will be an \( O(n \log n) \) algorithm. In order for this assumption to be true, the total size of the four partitions \( A_{(j-1)\epsilon}, B_{(j-1)\epsilon}, A_{j\epsilon}, B_{j\epsilon} \) must be small enough to fit in main memory. More formally:

\[
|A_{(j-1)\epsilon}| + |B_{(j-1)\epsilon}| \leq M \quad \text{for} \quad j = 1 \ldots [\frac{1}{\epsilon}] \quad (5.4)
\]

Figure 5.3b illustrates the two dimensional version of the algorithm. Given sets \( A \) and \( B \) containing points of the form \((x, y)\), the algorithm proceeds as follows: first, sort both files on the \( y \) coordinate and partition into two sets \( A_1, A_2 \) and \( B_1, B_2 \) by splitting on a single value of the \( y \) coordinate. Ideally, the partitioning should take place in such a way that \(|A_1| = |A_2| \) and \(|B_1| = |B_2|\). This can happen if both files have the same median
Given two one dimensional data sets, $A$ and $B$, and $\epsilon$:

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

Figure 5.4: One Dimensional MDC
value for each attribute. We apply the algorithm recursively in pairs of corresponding
partitions. This means that, for partitions $A_1, B_1$ ($A_2, B_2$ respectively), we have to apply
the one dimensional version of the algorithm just described. This involves sorting the
partitions on the $x$ coordinate and performing a linear scan, provided that elements in
the partitioning corresponding to ranges of size $2\epsilon$ fit in main memory. When this step
finishes, we still must take care of possible joining points close to the partition along the
$y$ coordinate. Only points within distance $\epsilon$ from both sides of the partitioning line are
candidates in this case and should be considered. We proceed to join pairs of partitions
$(A_1, B_2)$ and $(A_2, B_1)$ looking at candidate joining pairs only at distance $\epsilon$ from both
sides of the partitioning line. This step has to guarantee that the sets of points with
attribute values in ranges of length $2\epsilon$ fit in main memory. The area covered by each
of these ranges is $2\epsilon^2$. Generalizing this approach to $d$-dimensional spaces for data sets
involving $O(n)$ multidimensional points will give an $O(n \log^d n)$ [Ben80] algorithm.

Although, in the worst case, all points of a subpartition can be within distance $\epsilon$ from
a partitioning line, the number is much smaller under reasonable statistical assumptions.
Consequently, a variant of this algorithm could store a copy of all the points within
distance $\epsilon$ from a partitioning line while processing the partitions. Then we will have to
process only these smaller files to correctly compute the join, improving significantly the
algorithm's performance in most situations.

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

The $\epsilon$-KDB tree: A new indexing structure for the multidimensional join problem was proposed recently by Shim et al. [SSA97]. The $\epsilon$-KDB tree is intended to speed up the computation of hypercube intersections in main memory.

Given two multidimensional data sets and a distance $\epsilon$, the algorithm proceeds by choosing a dimension and sorting the data sets on this dimension. If both data sets are sorted already on a common dimension, no sorting is necessary. Then the algorithm proceeds to read the partitions corresponding to intervals of size $2\epsilon$ in the dimension of sorted order of both files into main memory and building the $\epsilon$-KDB structure on them. The structure is a variant of KDB trees [Rob81]. It offers a space decomposition scheme that facilitates tree matching since the boundaries of space partitions are canonical. That way, assuming both files have been sorted on a particular dimension, the algorithm can compute the join in time linear in the size of the input data sets by scanning the sorted data sets. In order for the time to be linear, however, the sum of the portions of both $A$ and $B$ in each $2\epsilon$ range along the chosen dimension must fit in main memory. If this is not the case, several problems arise. As new dimensions are introduced on which to perform partitioning, the algorithm must issue a complex schedule of non-sequential page reads from the disk. At each step, the algorithm has to keep neighboring partitions in main memory at the same time. The number of neighboring partitions is exponential in the number of dimensions used for partitioning. Assuming that $k$ dimensions are used for partitioning, in the worst case, we have to keep $2^k$ partitions from each file being joined in memory at all times. Since the pages holding the partitions are sequentially stored into disk, only two neighboring partitions can be stored sequentially. The rest of the relevant partitions for each step have to be retrieved by scheduling non-sequential I/Os.
Given two $d$-dimensional data sets, $A$ and $B$, and $\epsilon$:

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

Figure 5.5: The REPL Algorithm

5.3.1.3 Replication Approach (REPL)

The replication approach to the multidimensional join problem involves replicating entities and thus causing file sizes to grow. Algorithms based on the replication approach for the two dimensional problem have been proposed by Patel and DeWitt [PD96] and Lo and Ravishankar [LR96]. Here, we explore possible generalizations of these algorithms to higher dimensions. The underlying idea for these algorithms is to divide the two dimensional space into a number of partitions and then proceed to join corresponding partition pairs. It is possible that the size of a partition pair exceeds the main memory size and, as a result, the pair must be partitioned more finely. Points located in corresponding partitions form output tuples if they are found to be within $\epsilon$ distance of one another.

The algorithm as described generalizes directly to higher dimensions as shown in figure 5.5. We form a $d$ dimensional hypercube of side $\epsilon$ around each point in both multidimensional spaces and proceed in the same way. Assume that we wish to create $2^{jd}$ partitions of the space by dividing each of the $d$ dimensions into $2^j$ intervals, for some $j$. If points are uniformly distributed in the $d$ dimensional space, the fraction of
Figure 5.6: Fractions of replicated objects for various dimensionalities as a function of the probability that a hypercube is intersected in some dimension.

The problem of replication becomes worse as the dimensionality increases. It is evident that as dimensionality increases the fraction of objects replicated increases for a specific value of $\epsilon 2^j$. Intuitively this can be explained as follows: at dimensionality $d$ adding a new dimension imposes a partitioning of the $d + 1$ dimensional space by $2^j$ hyperplanes of $d$ dimensions. Consequently, the probability that objects are intersected by the new hyperplanes increases with $d$. Figure 5.7 shows an approximation of the probability that any particular point is replicated (according to the estimate of eq. (5.5)) as dimensionality increases for various values of $\epsilon$, keeping the number of space partitions constant and equal to ten thousand. The probability of replication remains relatively insensitive to the number of dimensions for small values of $\epsilon$. For larger values of $\epsilon$, the
probability of replication increases with dimensionality. We have to guarantee some degree of size balance across partitions; otherwise, it might be the case that a large portion of the file falls in one initial partition and extensive repartitioning is required.

There are two major drawbacks to approaches that introduce replication. The appropriate degree of partitioning of the data space is very difficult to choose unless precise statistical knowledge of the multidimensional data sets is initially available. Although having such knowledge might be possible for static multidimensional data sets, it is difficult and costly to obtain for dynamic data sets. Secondly, when points are relatively dense or $\epsilon$ is large, the amount of replication that takes place appears to be very large, and it becomes still larger as dimensionality increases.

5.3.1.4 Space Filling Curves Approach

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

Orenstein's Algorithm (ZC): Orenstein proposed an algorithm, that we call ZC, to perform joins of multidimensional objects [Ore91]. Starting with multidimensional
objects that are approximated by their minimum bounding hypercubes, the hypercubes are tested for intersections. For each pair of intersecting hypercubes, an object intersection test is performed. The algorithm is based on z-curves and their properties. Z-curves reflect a disjoint decomposition of the space and ZC relies on the following property of z-curves to detect intersections: Two approximations of multidimensional objects intersect if and only if the z-value which has the shorter binary representation is a prefix of the z-value with the longer binary representation.

The algorithm imposes a recursive binary splitting of the space up to a specific granularity. Each approximated entity is placed in a space division that fully encloses it. Orenstein [Ore89] presents an analysis of the implication of this decomposition scheme on range query performance, and, in subsequent work [Ore91], presents the performance of the multidimensional join algorithm.

This algorithm can be applied to the multidimensional join problem that we address in this chapter. In our context, each multidimensional point is approximated with a d dimensional hypercube of side $\epsilon$. For each multidimensional point, the z-curve value (ZV) at dimensionality $d$ is computed. The ZV is a variable length bit string. Shorter bit strings correspond to larger space partitions in the recursive binary decomposition of the space. As dimensionality increases, the processor time to compute the ZV, as well as the number of bits required to store it increases. We assume that, in a preprocessing step, the ZV associated with each point is computed to some specified precision.

Each data set is scanned and the ZV of each hypercube is transformed to the ZV of the smallest space partition that contains it. The transformation involves setting to zero a number of least significant bits of ZV, depending on the size of the space partition that contains the hypercube. Both data sets are then sorted into non-decreasing order of ZV values. Then the algorithm proceeds to merge the two data sets using a stack per data set. At each step, the smaller ZV is selected and processed by comparing it to the ZV at the top of the stack. The algorithm is shown in figure 5.8. The distance predicate is evaluated between pairs of elements whose hypercubes intersect to determine whether the point pair belongs in the final join result.

The algorithm as proposed by Orenstein [Ore91] allows for the decomposition of
Given two $d$-dimensional data sets, $A$ and $B$, and distance $\epsilon$:

For each data set:

Scan the data set and transform the ZV of the center of the hypercube in the space partition that encloses it.

Sort the data set into nondecreasing order of ZV's.

Initialize to empty a stack per data set ($S_A$ and $S_B$)

Loop until either $A$ or $B$ is empty

Let $x_A$ be the next ZV from $A$ and $x_B$ from $B$

if ($x_A \leq x_B$) $IT \leftarrow A$; $OTHER \leftarrow B$

else $IT \leftarrow B$; $OTHER \leftarrow A$

if $x_{IT}$ is a prefix of $top(OTHER)$ then push($x_{IT}$, $OTHER$)

else

loop until either $OTHER$ is empty or $x_{IT}$ becomes a prefix of $OTHER$

pop($OTHER$)

end loop

evaluate the distance predicate for each pair of overlapping hypercubes.

end loop

Figure 5.8: The ZC Algorithm
multidimensional objects into a number of pieces. This is useful when the sum of the volumes of hypercubes for each component is significantly less than the volume of the hypercube around the whole object. In our case, however, we deal only with hypercubes around points so decomposition is not beneficial. Also, decomposing not only increases database size but also introduces the necessity of a duplicate elimination phase in the algorithm.

5.3.2 Algorithms That Use Preconstructed Indices

The best known spatial join algorithm for R-trees is the one proposed by Brinkhoff et al. [BKS93]. We have extended it to apply to multidimensional point sets indexed with $R^*$-trees [BKSS90]. The $R^*$-tree join algorithm is based on an index sweeping process. The algorithm proceeds top-down sweeping index blocks at the same level in the two indices. At a specific level, the pairs of overlapping descriptors are identified and, at the same time, the hyperrectangles of their intersections are computed also. This information is used to guide the search in the lower levels, since descriptors not overlapping the hyperrectangle of intersection of their parents need not be considered for the join. The algorithm uses a buffer pinning technique that follows a greedy approach trying to keep relevant blocks in the buffer in order to minimize block re-reads. If the indices do not have the same height, the algorithm proceeds as described above up to a certain point and then degenerates into a series of range queries.

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

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

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

5.4 Multidimensional Spatial Join (MSJ)

To perform the join of two multidimensional data sets, A and B, we may also use a generalization of the Size Separation Spatial Join algorithm ($S^3$J) [KS97] that was presented in chapter 4. The generalization is called Multidimensional Spatial Join (MSJ) [KS98].

The $S^3$J algorithm makes use of space filling curves to order the points in a multidimensional space. We assume that the Hilbert value of each multidimensional point in d dimensions is computed to $dL$ bits of precision where $L$ is the maximum number of levels of size separation. We consider the Hilbert value computation a preprocessing step of this algorithm. For two d dimensional data sets, A and B, and given a distance $\epsilon$, we impose a dynamic hierarchical decomposition of the space into level files. We scan each data set and place each multidimensional point $(x_1, x_2, \ldots, x_d)$, in a level file $l$, determined by

$$l = \min_{1 \leq i \leq d} ncb\left(x_i - \frac{\epsilon}{2}, x_i + \frac{\epsilon}{2}\right)$$  \hspace{1cm} (5.6)

where $ncb(b_1, b_2)$ denotes the number of most significant common bits in bit sequences $b_1$ and $b_2$. This corresponds to the placement of the approximated multidimensional point in
the smallest subpartion of the multidimensional space that fully encloses it. The Hilbert value, $H$, of each multidimensional point is transformed to the maximum Hilbert value of the space partition that encloses it at level $l$. This transformation can be achieved by setting to one the $(L - l)d$ least significant bits of $H$. This transformation is necessary to assure that we don't miss any possible joining pairs. By transforming the Hilbert value of a multidimensional point, to the maximum Hilbert value of the space partition that encloses it at label $l$, we make sure that the multidimensional point will remain in the buffer pool for as long as needed. Each level file is then sorted into nondecreasing order of Hilbert values.

The decomposition of the multidimensional space achieved this way provides a flexible way to perform the multidimensional join [KS97]. Each subpartition of a level file has to be matched against the corresponding subpartitions in the corresponding and each higher level file of the other data set. That way, in the worst case, we need to keep in memory as many subpartitions for each data set as there are level files. Figure 5.10 presents the algorithm. Both data sets are scanned and partitioned into level files. At the same time Hilbert value transformation takes place. All level files are sorted on the Hilbert value. Finally, a multi-way merge of the level files takes place.

Figure 5.9 illustrates the merge phase of the algorithm. Two files ($F_1$ and $F_2$) have been partitioned into two level files each. At first, the algorithm issues a read of one partition from each level file in main memory, thus partitions $P_1, P_2, P_3, P_4$ will be read. The minimum starting Hilbert value over all partitions ($H_{ms}$) and also the minimum ending value ($H_{me}$) are computed. Corresponding entries between $[H_{ms}, H_{me}]$ can be processed in main memory. Partitions that are entirely processed ($P_3$ in figure 5.9) are dropped from the buffer pool and $H_{me}$ is updated. Processing can continue by replacing processed partitions from the corresponding level files (read in $P_7$ in figure 5.9) and advancing $H_{me}$ as needed, until all level files are processed.

In separating the points in each data set into level files, we may use any subset of the dimensions. The number of dimensions used to separate the input data sets to level files affects the occupancy of each level file. In a $d$ dimensional space, for level file $l$, there are $2^d$ space partitions. Each non-empty partition will have to be memory resident at
Given two $d$-dimensional data sets, $A$ and $B$, and $\epsilon$ distance predicate:

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

Figure 5.10: The MSJ Algorithm

...
5.5 Experimental Evaluation

In this section, we present an experimental evaluation of the performance of MSJ relative to some of the algorithms described in the previous sections for joining multidimensional data sets. All algorithms were implemented in the C programming language under the AIX operating system. Our implementation of MSJ is approximately 600 lines of C code.

5.5.1 Description of Data sets

For our assessment of the performance of the multidimensional join algorithms, we used both synthetic and real data sets of various dimensionalities. Since the size of each record grows with the number of attributes (dimensions), the overall file size for the fixed number of points increases with the number of dimensions. We choose to keep the number of multidimensional points constant in spaces of different dimensions. An alternative would be to keep the total file size constant by reducing the total number of points as dimensionality increases. However, this would create very sparsely populated multidimensional spaces and the performance of multidimensional joins for increasing values of $c$ would be difficult to assess, unless very large file sizes were used.

Table 5.1 presents the data set sizes in terms of total number of points and total file sizes in bytes at the dimensionalities used in our experiments. We keep the buffer pool size constant (2MB) for all experiments.

We perform four series of experiments involving synthetic and real data sets. For each series of experiments, we report two sets of results. In one, we keep $c$ constant and increase the dimensionality of the data set. In the other, we keep the dimensionality of the data set constant and increase the value of $c$.

The first series of experiments involved multidimensional self joins between uniformly distributed data sets. The data set used for this experiment has characteristics $D1$. The second experiment involved two data sets, each having characteristics $D2$, containing uniformly distributed multidimensional points generated with different pseudo-random

\[\text{Table 5.1 presents the data set sizes in terms of total number of points and total file sizes in bytes at the dimensionalities used in our experiments. We keep the buffer pool size constant (2MB) for all experiments.}\]

\[\text{We perform four series of experiments involving synthetic and real data sets. For each series of experiments, we report two sets of results. In one, we keep } c \text{ constant and increase the dimensionality of the data set. In the other, we keep the dimensionality of the data set constant and increase the value of } c.\]

\[\text{The first series of experiments involved multidimensional self joins between uniformly distributed data sets. The data set used for this experiment has characteristics } D1.\]

\[\text{The second experiment involved two data sets, each having characteristics } D2, \text{ containing uniformly distributed multidimensional points generated with different pseudo-random}\]

\[\text{The use of congruential random number generators to create the multidimensional vectors has a distinctive statistical behavior and should be avoided. The resulting multidimensional space has the characteristic that its elements lie mainly on parallel planes [Mar68]. In order to avoid this phenomenon, we use a method to create the multidimensional vectors suggested by Fishman [Fis73].}\]
number seeds. Although these experiments involving uniformly distributed data sets offer intuition about the relative performance of the algorithms, it is highly likely that real multidimensional data sets will contain clusters of multidimensional points. These clusters will correspond to groups of entities with similar characteristics. For this reason, in the third series of experiments, we generated multidimensional data sets containing clusters of multidimensional points, and we evaluated the performance of the algorithms using the resulting data sets, which again have characteristics D2. A cluster was generated by initializing a kernel in the center of the data space and distributing the points around the cluster kernel. Points outside the unit hypercube were clipped. The coordinates of each multidimensional point were generated using an exponential distribution with mean 0.5. The sign of each coordinate was uniformly chosen, and the point was positioned relative to the kernel by adding the point coordinates and the cluster coordinates.

Finally, the fourth series of experiments involved actual stock market data on price information collected for 501 companies. We applied a Discrete Fourier Transform (as suggested by Faloutsos [FRM94]) to transform the time series information into points in a multidimensional space. Using a period of ten days, we extracted several time series from the sequence of prices for each specific stock, obtaining 84,640 multidimensional points. The resulting data set had characteristics D3.
### Table 5.2: Experiments performed and characteristics of data sets involved in each experiment. The % buffer column reports the buffer space available to each experiment as a percentage of the total size of the data sets joined, for various dimensionalities.

| Experiment number | Kind of Operation | Characteristics of Data Sets | % buffer  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Self Join</td>
<td>D1-uniform</td>
<td>d=3/4</td>
</tr>
<tr>
<td>2</td>
<td>Non Self Join</td>
<td>D2-uniform</td>
<td>66%</td>
</tr>
<tr>
<td>3</td>
<td>Self Join</td>
<td>D2-clustered</td>
<td>66%</td>
</tr>
<tr>
<td>4</td>
<td>Self Join</td>
<td>D3-actual</td>
<td>30.8%</td>
</tr>
</tbody>
</table>

5.5.2 Experimental Results

5.5.2.1 Experiments with Algorithms Not Based on Preconstructed Indices

Our experiments are summarized in table 5.2. The results of these four experiments are presented in figures 5.11, 5.13, 5.14 and 5.15, respectively. Figure 5.11 presents the performance of multidimensional self joins (experiment 1). Note that for dimensionality $d = 3$, the response time for MSJ is factors of 3, 6 and 20 lower than that of ZC, REPL and nested loops, respectively. At dimensionality $d = 20$, the corresponding factors are approximately 2, 3 and 30. Figure 5.11(a) presents the performance of the algorithms for $\epsilon = 0.011$. As dimensionality increases, the response time of MSJ increases due to increased sorting cost, since the buffer space available to the sort holds smaller and smaller fractions of the data sets. The processor cost increases only slightly with dimensionality, since the size of the join result does not change much. At low dimensionality the size of the join result is a little larger than the size of the input data sets and it decreases to become equal to the size of the input data sets as dimensionality increases. At higher dimensions, a hypersphere of fixed radius inscribes lower percentages of the total space, and the probability for a point to match with something more than itself drops rapidly.

Figure 5.11(b) presents the performance of self joins for increasing values of $\epsilon$ at dimensionality $d = 12$. The performance of MSJ appears almost constant, since processor
time increases only slightly for these values of \( \epsilon \), as shown in figure 5.12(b). (The processor time increase corresponds to the small increase in the Join phase of MSJ in figure 5.12). I/O time for MSJ remains almost constant for the range of \( \epsilon \) values of figure 5.11(b). For REPL, response time increases due to the increase of replication with \( \epsilon \). Similarly, increased replication causes an increase in processor time. For nested loops, processor time increases with \( \epsilon \), since the multidimensional sweep in main memory checks more candidate pairs. The join result size for experiment 1 does not change much for the range of \( \epsilon \) values presented.

Although the I/O behavior of MSJ and the ZC algorithm is the same, there are additional processor costs for the ZC algorithm. Figure 5.12(a) presents the portions of time spent in the various phases of the algorithms. The main difference between MSJ and the ZC algorithm is that the sweep process in main memory is data driven for MSJ but partition driven for ZC. ZC relies on the prefix property of the z-curve to perform the join. Candidates have to be generated from the stack each time the prefix property of the curve is violated. Violation of the prefix property takes place each time the curve crosses boundaries between different space partitions. Since partitions are seldom full and thus are collapsed together in physical pages, this leads to a large amount of data movement in and out from the stacks, as well as plane sweep operations, which constitute an additional processing cost for the algorithm, as is evident from figure 5.12(a). Moreover, ZC requires data structure manipulations on the stacks and prefix evaluations for each multidimensional point of the data sets being joined.

For REPL, the amount of replication during the partitioning phase increases with dimensionality (as indicated by eq. 5.5) and this increases both processor and I/O cost. Processor cost increases because, by introducing replication, more points are swept. In addition, a duplicate elimination phase has to take place at the end of the algorithm, and this involves a sort of the result pairs. Finally, the response time of nested loops increases with dimensionality since relatively less buffer space is available to the operation.

Figure 5.13 presents the performance of multidimensional joins between two uniformly distributed data sets of type D2 generated with different seeds (experiment 2). Figure 5.13(a) presents the response times as dimensionality increases for \( \epsilon = 0.05 \). The join
(a) Increasing dimension for \( \epsilon = 0.011 \)

(b) Increasing \( \epsilon \) for \( d = 12 \)

Figure 5.11: Performance of self joins on data sets D1

(a) Increasing dimension for \( \epsilon = 0.011 \)

(b) Increasing \( \epsilon \) for \( d = 12 \)

Figure 5.12: Portion of time spent at different phases of the algorithms
(a) Increasing dimension for \( \epsilon = 0.05 \)  
(b) Increasing \( \epsilon \) for \( d = 12 \)

Figure 5.13: Performance of multidimensional joins between two distinct uniformly distributed data sets

(a) Increasing dimension for \( \epsilon = 0.006 \)  
(b) Increasing \( \epsilon \) for \( d = 8 \)

Figure 5.14: Performance of multidimensional joins between two distinct clustered data sets
distance predicate is relatively large and, in the case of three dimensions, the join result is large as well. For MSJ, we observe a small decrease in execution time as we move from three to eight dimensions which can be explained by taking into account the size of the join result. For dimensionality \( d = 8 \), the size of the join result is much smaller, and this explains the difference in execution time. As dimensionality increases, the size of the join result is smaller, but sorting costs increase since the ratio of the buffer pool size to the total input size becomes smaller, and this accounts for the increase in total execution time of MSJ. For the ZC algorithm, we observe a similar behavior. Sorting costs increase as dimensionality increases, and, as with the self join, a larger fraction of processor time is needed to compute the join by ZC relative to MSJ.

For REPL at dimensionality \( d = 3 \), producing and sorting the join result takes a large fraction of the total execution time. As dimensionality increases, I/O time (due to replication and sorting) and processor time (due to sorting and generating additional candidates) increase as well. Replication increases faster than in figure 5.11(a) because the join predicate is larger (see eq. (5.5)). For nested loops, I/O time as well as processor time increases due to the decreasing ratio of buffer space to file size available to the operation, and accounts for the increase in execution time as dimensionality increases.

Figure 5.13(a) presents the response time of all algorithms for increasing values of \( \epsilon \) at dimensionality \( d = 12 \). For REPL, execution time increases with \( \epsilon \) due to increased
I/O time and processor time. The increase is sharper as $\epsilon$ increases because larger $\epsilon$ values mean that more hypercubes cross boundaries, so that more replication occurs. For algorithms based on either space filling curves or nested loops, the observations remain the same.

Comparing figure 5.13, to figure 5.11 we see that the performance differences among the algorithms are generally larger for the join of distinct data sets relative to for self joins. The response time of ZC is about twice that of MSJ quite consistently as $\epsilon$ and $d$ are varied. The response time of REPL is about 12 times that of MSJ for all dimensionalities when $\epsilon = 0.05$. It ranges from 4 to 30 times as big as $\epsilon$ goes from .01 to .10 for dimensionality 12. Finally, Nested Loops requires 20 to 40 times longer than MSJ over the range of dimensionalities for $\epsilon = 0.05$. Figure 5.13 shows that as $\epsilon$ increases, the difference in performance between the space filling curve based methods (MSJ and ZC) and the other methods (REPL and Nested loops) increases substantially.

Figure 5.14 presents the response time of the algorithms for experiment 3, which involves two data sets containing points that are clustered rather than uniform. The trends in performance for all algorithms for increasing dimensionality are similar to those in the previous experiments (see figure 5.14(a)). Figure 5.14(b) presents the response time of the algorithms for increasing values of $\epsilon$ at dimensionality $d = 8$. For all the algorithms, response time increases more rapidly with $\epsilon$ than in experiment 1 (for a similar range of $\epsilon$ values). Due to clustering, the increase in the size of the join result is larger, and, as a result, the processor time needed to compute the candidate and actual pairs increases.

Figure 5.15 presents the performance of the algorithms for experiment 4, which involves real stock market data. We employ a multidimensional join operation which reports only the total number of actual joining pairs. (We do not materialize the full join results, due to their size.). In figure 5.15(a), we present the response time of the algorithms for $\epsilon = 0.03$ as dimensionality increases. For nested loops and REPL, the basic observations are consistent with those from previous experiments. Both algorithms that use space filling curves have increased response times due to their sorting phase as dimensionality increases. However, processor time drops due to the smaller join result size with increasing dimensionality. Both algorithms are processor bound for this
Table 5.3: Summary of approximate response time ratios of other algorithms to MSJ

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Exp</th>
<th>d varies</th>
<th>ε varies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{\text{Nested}}/R_{\text{MSJ}}$</td>
<td>E1</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>E3</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>E4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>$R_{\text{REPL}}/R_{\text{MSJ}}$</td>
<td>E1</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>E3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>E4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.5</td>
<td>2.5</td>
</tr>
<tr>
<td>$R_{\text{ZC}}/R_{\text{MSJ}}$</td>
<td>E1</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>1.3</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>E3</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>E4</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 5.3: Summary of approximate response time ratios of other algorithms to MSJ

experiment, and this explains the smoother increase in response time as dimensionality increases.

Figure 5.15(b) presents the response time of the algorithms at dimensionality $d = 4$, for increasing values of $\epsilon$. All algorithms appear to be processor bound, and the increase in the join result size accounts for the increase of response times for all algorithms.

Table 5.3 presents a summary of approximate response time ratios between other algorithms and MSJ as observed in our four experiments. The results are reasonably consistent over the ranges of $d$ and $\epsilon$ that we explored. The ZC algorithm had response times between 1.3 and 3 times longer than MSJ over the range of experiments. The REPL algorithm showed more variability in its relative performance, with ratios ranging from 2 to 30 in various cases. Finally, the response times of nested loops were 4 to 100 times larger than MSJ's over the range of cases tested.
5.5.2.2 Experiments with Algorithms Based on Preconstructed Indices

The experimental results presented for algorithms that don't require preconstructed indices suggest that approaches based on space filling curves, and specifically MSJ, are effective in solving the multidimensional join problem. We also investigate the performance of MSJ in comparison to the algorithms that utilize preconstructed indices.

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

Figure 5.16(a) presents the performance of both MSJ and RTJ for self join of a data set having characteristics D1 as dimensionality increases. For MSJ the observations remain exactly the same as those pertaining to figure 5.11. The performance of RTJ deteriorates as dimensionality increases. As dimensionality gets larger, the overlap between R-tree index entries and leaf entries increases. As a result, the number of pages that have to be pinned in the buffer pool is likely to increase as well. Since the size of the buffer pool is kept constant for varying dimensionalities for both algorithms, the number of page rereads that RTJ has to schedule is expected to increase with dimensionality, and this
explains the deterioration in performance. The performance of RTJ is very sensitive to the amount of buffering available to the operation. Since the overlap in the R-tree index is expected to increase with dimensionality, the sensitivity gets stronger as dimensionality increases. Figure 5.16(b) presents the performance of both MSJ and RTJ for increasing epsilon and dimensionality $d = 12$. Both algorithms require more processor time due to the increasing number of join tests for increasing values of epsilon. However, the performance of the RTJ is worse than that of MSJ, since it requires a larger number of I/Os.

5.6 Conclusions about MSJ

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

Nested loops is applicable in all circumstances, but has a computational complexity that matches the complexity of the multidimensional join problem. Its performance is poor due to redundant processor and IO work, which, for a variety of data distributions, can be avoided. Introducing replication must be done with care, particularly in multiple dimensions. Replication always leads to additional processor and IO work. Our experimental results indicate that algorithms based on space filling curves, and specifically MSJ, seem promising for computing multidimensional joins across a range of dimensionalities, even if multidimensional R-tree indices already exist for the data sets involved.
Chapter 6
Conclusions

In this thesis, we presented algorithms for efficiently executing spatial and multidimensional join operations. We presented Filter Trees for executing joins in the static case, when indices are available for the data sets involved in the spatial join operation. We then presented the $S^3J$ algorithm for the spatial join problem in the dynamic case, when indices are not available for the data sets being joined. Finally, we proposed a generalization of the $S^3J$ algorithm to higher dimensional spaces, MSJ. In all cases, we have presented experimental results, comparing our algorithms with previous approaches and we showed the performance benefits of our approach. There are several directions for extending the work.

6.1 Refinements

For Filter Trees other types of queries, particularly the nearest neighbor query, should be studied. The field of spatial data mining has generated interest in spatial access methods and their connection to data mining. Filter Trees, due to their recursive decomposition of the space seem suitable for computing the distribution of the data at a fine level inside the cells. Thus, exploration of Filter Trees in the spirit of Wang, Yang and Muntz [WYM97] could prove a fruitful area of research.

The Size Separation Spatial Join algorithm, as well as its multidimensional extension, MSJ, merits further exploration for their applications to data mining of times series, image, audio and video data types. Alternative space decompositions, in the spirit of the Vantage Point Tree [Yia92], during the execution of the refinement step might prove ben-
eficial in reducing the processor overhead of this step. In addition, alternative semantics of the multidimensional join should be investigated, like the all pair nearest neighbor query (find all pairs of points closest to each other). We expect the all pair nearest neighbor query to be very important in the context of multimedia similarity queries. It would be interesting to see whether the Filter Tree nearest neighbor query and $S^3J$ can be efficiently combined to provide a solution to the all pairs nearest neighbors query.

Finally, another interesting area for future work is the study of Filter Trees as well as $S^3J$ in the context of parallel database architectures. Some initial thoughts in that direction are presented in appendix A.

6.2 Integration Into Practice

We expect both Filter Trees and $S^3J$ to be integratable as an extension into commercial Object Relational Database Management Systems. For example, they can be implemented as a DataBlade for the Informix universal server or as an extender for the IBM DB2 V5 universal database. Such an implementation is possible, since, through the use of Hilbert curves on the data space, the algorithms proposed in this thesis transform the input data sets, into totally ordered relations. So the adoption of our algorithms does not require structural changes in the ORDBMS. Only the code implementing the algorithms has to be integrated.
Appendix A

Parallelization

We believe that the underlying concept of the algorithms presented in this thesis, the effective linearization of the data space though the use of space filling curves, makes them suitable for efficient parallelization. We illustrate the methodology that one could follow in order to parallelize the $S^3J$ algorithm. A similar methodology would apply in the case of Filter Trees and MSJ. We focus on shared nothing parallel database architectures, due to their promise for scalability and their efficient usage of commodity hardware.

There are two issues that any parallel algorithm for join execution should face: (a) evenly distribute the load across the nodes of the architecture and (b) limit the amount of data movement between nodes. We discuss how a parallel design of $S^3J$ could efficiently handle these issues, both of which affect the execution time of the algorithm.

As is evident from our presentation of $S^3J$ and MSJ in chapters 4 and 5, the spatial join problem, for several data sets, can be processor bound. Thus, the notion of "load" for a parallel spatial join algorithm, should include both processor and disk time, unlike for parallel relational join algorithms. Assume that the parameters of the problem are perfect powers of two. The discussion generalizes to other cases as well. For ease of exposition, assume that the number of nodes in the parallel architecture is $M = 4^m$, the total number of cells in the last level of the $S^3J$ decomposition of the space is $N = 4^n$. In addition, we choose a declustering unit, which represents the number of cells from the data space that will be assigned to a node at each time. Let the declustering unit be, $K = 4^k$. The value of $K$ is of great importance to the algorithm. In a sense, $K$ determines the extent to which load balance will be achieved across the nodes. The smaller the value
is, the finer the partitioning of the data across nodes becomes. This finer partitioning however does not come for free. Assuming specific values of $M$ and $N$ and that all $n + 1$ levels in the $S^3J$ decomposition exist, the levels from 0 to $n - k - m + 1$ have to be replicated across all nodes, in order to assure correctness. Thus a tradeoff exists between the value of $K$ and the amount of information that gets replicated across all nodes. In practice, however, the upper levels of a $S^3J$ decomposition are highly unlikely to be present. Moreover, even if the higher levels are present, as suggested by our analysis of Filter Trees presented in Chapter 3, the fraction of objects that they will contain is very small. Consequently they could be easily accommodated into main memory. Thus, in practice, small values of $K$ can be used without the necessity to replicate a significant amount of information.

An additional parameter that should be considered for the choice of $K$ is the expected contribution of each declustering unit to the total join result. Assume that the contribution of each declustering unit, $D_i$, to the total join cost, $D$, is $P_i + I_i$, where $P_i$ is the processor cost to produce the join result in the unit $D_i$ and $I_i$ is the cost due to movement of the data in $D_i$. We wish to balance the $D_i$'s across the nodes such that each node receives $\frac{D}{M}$ of the load. Correct estimation of each $D_i$ consists of estimating both $I_i$ and $P_i$. The estimation of $I_i$ is less troublesome, since a pass over the data can give statistical information about the data distribution we are dealing with. The estimation of $P_i$, however, is still an open problem. Previous work for estimating the selectivities of spatial joins made several assumptions about the underlying data distributions which don't always hold in practice [BF95]. Several options exist for distributing the $D_i$'s across nodes. We can either apply a round robin algorithm or use the list scheduling heuristic that can provably approximate the optimal placement within a factor of two [Gra69][DNSS92].

Once the data are distributed across the nodes, each node can apply the $S^3J$ algorithm to the data stored locally and produce its results. Notice that no duplicate elimination is necessary using this approach.

An interesting solution to the parallel spatial join problem has been presented by Shafer and Agrawal [SA97]. It would be interesting to compare it with the parallel version of the spatial join algorithms proposed in this thesis.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


[NH94] Raymond T. Ng and Jiawei Han. Efficient and Effective Clustering Methods for Spatial Data Mining. *Proceedings of the 20th International Conference on VLDB*, pages 144–155, September 1994.


