EFFICIENT DATABASE SUPPORT FOR SPATIAL APPLICATIONS

By

Jignesh M. Patel

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

at the

UNIVERSITY OF WISCONSIN – MADISON
1999
Abstract

This thesis concerns efficient support of geo-spatial applications on large volumes of spatial data. This thesis examines efficient algorithms for evaluating two important spatial operations: the spatial join and the spatial aggregate. A spatial join combines two spatial data sets based on some spatial relationship between the elements in the two data sets. For example, map overlap, which requires combining two maps to produce a third, is a spatial join. A spatial aggregate summarizes the input by applying a spatial aggregate function. An example of spatial aggregation is locating the closest airport to a point on the map. Both the spatial join and the spatial aggregate operations are very expensive to compute. Consequently, efficient algorithms for both these operations are critical to the overall performance of a spatial database system. The first part of this thesis presents a new algorithm for evaluating a spatial join, and compares its performance with other spatial join algorithms. This part of the thesis also presents an algorithm for evaluating the spatial aggregate operation. This spatial aggregate algorithm uses a spatial index to speed up the evaluation.

The second part of this thesis focuses on using parallelism to evaluate spatial queries. Two issues are considered here: how to decluster spatial data in a parallel database system, and how to use parallelism to efficiently evaluate spatial operations. Various spatial declustering strategies, which are based on partitioning the space in which the spatial objects lie and mapping the space partitions to nodes, are presented and evaluated. Then, based on these declustering strategies, a design space for parallel spatial join algorithms is presented. Analytical models and results from an actual implementation show that the performance of these algorithms is very sensitive to the underlying characteristics of the spatial data. Finally, this thesis presents an algorithm for evaluating the spatial aggregate operation in a parallel environment.
Acknowledgements

The harvesting of this thesis is the result of careful sowing and nurturing by many people. The first and most important seed was planted by my parents. Besides providing the loving and caring fatherly touch, my father also provided all his children with a role-model right in our home. My mother did a terrific job of raising four children. She took great interest in our schooling right from kindergarten, and spent countless hours teaching us at home after school. This thesis would not have been possible without the support of my parents.

Here at Wisconsin, I've had the pleasure of working with my advisor David DeWitt. David infects everyone with his passion for work and his enthusiastic zeal, and though many times he puts on a very hard-to-play attitude, he is in fact a very caring and considerate person. Besides being very successful with his career, David has come close to perfecting the art of successfully managing his career, his personal hobbies, and his family life. I'd be really happy if I can mimic just a small part of that balancing skill. Besides David, Mike Carey and Mary Vernon have been very helpful and encouraging during my entire stay at Wisconsin. In fact, I wrote my first conference paper with Mike and Mary, torturing both of them with my naive technical writing skills. Nevertheless, both Mike and Mary never complained, and cheerfully pointed out all my mistakes. I would also like to thank Raghu Ramakrishnan for taking a very keen interest in guiding my career and always taking the time to listen to me patiently as I mumbled confusingly about my job options. Jeff Naughton and Teresa Adams deserve special thanks for the many useful discussions that I've had with them about my thesis. I'm also very thankful to David Maier for taking the time to be on my defense committee, and for reading every single work of my thesis carefully pointing out various improvements.

I would like to thank all my fellow-members of the Paradise team for being so supportive of my dissertation research. I would specially like to thank Jie-Bing Yu for patiently answer my questions about the lower layers of SHORE. I've bugged Joseph Burger countless times with hardware problems that he always fixed cheerfully. Nancy Hall has always been willing to take the time and track down and fix any problems that I discovered in SHORE. Navin Kahra and Biswadeep Nag also deserve special thanks for the many fruitful discussions that I've had with them, and also for reading and commenting on various drafts of my papers and reports. Karthikeyan Ramasamy deserves special thanks for always answering questions that I had on NT, Java, and networking, and for keeping us updated with the latest happenings on Wall Street. I would also like to thank the members of the CS lab staff for doing an excellent job in keeping the machines running in the department. Lorene Webber and Marie Johnson deserve thanks for keeping the department running so efficiently.

Life in the cold frozen winters of Madison would have been very unfriendly and uncomfortable if it hadn't been for all my friends here. Kurt Brown, with whom I shared an office for three years made the office lively with his great sense of humor and enchanting personality. The racquet bouts that we had ensured that we kept the winter fat off us. During my long stay at Madison, I had the good fortune of meeting Praveen Seshadri and Ranjani Ramamurthy with whom I spent many pleasant evenings eating dinner (cooked by them), discussing movies, cars and cultural shock of being in the US, and worrying about my marital future. I deeply cherish their friendship, and the sweet memories of the time that we have spent together. And then there is this guy who goes
by the name of Babak Falsafi, whom I first encounter when he was playing the role of the TA for my computer architecture course. As a TA, I found that he was very miserly when it came to grading (his students at Purdue are in big trouble), but as a housemate and a friend, we had a great time arguing spiritedly over various issues, watching movies, experimenting with cuisine from all over the world, and playing tennis and basketball. Talking of great friends, my dear friend Natassa Ailamaki deserves special thanks for the many pleasant socials that we have shared together, and for very memorable trips to Chicago and Milwaukee (keep that ID stapled to you at all times, my friend!). I am also very thankful to Stefanos Kaziras and Angeliki Baltoyianni for the many movies that we saw together, the biking and inline-skating excursion that we had through remote parts of Madison, the discussions in the coffee shops, and for the many wonderful dinners that we have shared together. I’d also like to thank T. N. Vijaykumar for the entertaining discussions that we had in our living room, and Satish Chandra for sharing his interest in music and literature with me.

Finally, I would like to thank Neha for being the person that I've always dreamt of meeting, with whom I can contemplate spending more than a lifetime. I had nearly given hope before I met her.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Evolution of Database Systems</td>
</tr>
<tr>
<td>2</td>
<td>The Spatial Partitioning Function</td>
</tr>
<tr>
<td>3</td>
<td>Spatial Sort Merge Example</td>
</tr>
<tr>
<td>4</td>
<td>Spatial Partitioning Skew</td>
</tr>
<tr>
<td>5</td>
<td>Spatial Partitioning Function using Tiles</td>
</tr>
<tr>
<td>6</td>
<td>Spatial Partitioning Function Alternatives: Tiger Road Data</td>
</tr>
<tr>
<td>7</td>
<td>Replication Overhead: Tiger Road Data (16 Partitions)</td>
</tr>
<tr>
<td>8</td>
<td>Replication Overhead: Sequoia Polygon Data(16 Partitions)</td>
</tr>
<tr>
<td>9</td>
<td>TIGER Data: Join Road with Hydrography</td>
</tr>
<tr>
<td>10</td>
<td>TIGER Data: Join Road with Rail</td>
</tr>
<tr>
<td>11</td>
<td>Clustered TIGER Data: Join Road with Hydrography</td>
</tr>
<tr>
<td>12</td>
<td>Clustered TIGER Data: Join Road with Rail</td>
</tr>
<tr>
<td>13</td>
<td>R-tree-based Algorithm, TIGER Data: Join Road with Hyd. Clustered and non-clustered (NC) scenario</td>
</tr>
<tr>
<td>14</td>
<td>Indexed Nested-Loops Algorithm, TIGER Data: Join Road with Hyd. Clustered and non-clustered (NC) scenario</td>
</tr>
<tr>
<td>15</td>
<td>PBSM Algorithm, TIGER Data: Join Road with Hyd. Clustered and non-clustered (NC) scenario</td>
</tr>
<tr>
<td>16</td>
<td>Sequoia Data Set</td>
</tr>
<tr>
<td>17</td>
<td>R-tree-based Algorithm, Sequoia Data: Join Polygons with Islands</td>
</tr>
<tr>
<td>18</td>
<td>Indexed Nested-Loops Algorithm, Sequoia Data: Join Polygons with Islands</td>
</tr>
<tr>
<td>19</td>
<td>PBSM Algorithm, Sequoia Data: Join Polygons with Islands</td>
</tr>
<tr>
<td>20</td>
<td>Comparison of the Join Algorithms with indices, TIGER Data (Join Road with Hydrography)</td>
</tr>
<tr>
<td>21</td>
<td>Comparison of the Join Algorithms with indices, TIGER Data (Join Road with Rail)</td>
</tr>
<tr>
<td>22</td>
<td>Buffer Zone for Probing the Index</td>
</tr>
</tbody>
</table>
23 Farthest Query, Buffer Zone for Probing the Index ........................................ 29
24 Farthest Query, Buffer Zone Represented by Rectangles .............................. 29
25 Operator Tree for Aggregate Evaluated with a Nested Loops Join ................. 30
26 Join with Aggregate Operator Tree ............................................................. 30
27 Find the Closest Drainage Feature to the Selected Cities ............................ 31
28 Declustering of Polygon Data ...................................................................... 35
29 Declustering Highways .................................................................................. 35
30 Data Distribution Skew ............................................................................... 36
31 Tiling to Reduce Data Distribution Skew ...................................................... 36
32 Spatial Distribution Skew: DCW Drainage Data ........................................... 37
33 Replication Overhead: DCW Drainage Data ................................................ 37
34 Spatial Distribution Skew: Sequoia Landuse Data Set ................................ 38
35 Replication Space Overhead with the Sequoia Landuse Data Set ................. 38
36 Replication of Tuples with the Sequoia Landuse Data Set ........................... 38
37 Declustering Using Spatial Surrogates ......................................................... 38
38 Declustering Overhead of the Sequoia Landuse Data Set Using Partial Spatial Surrogates ........................................................... 39
39 Need for Duplicate Elimination .................................................................... 41
40 Design Space of Spatial Join Algorithms ...................................................... 41
41 Join Using Partial Spatial Surrogates For Redeclustering (Algorithm A—Shadow Join) .................. 42
42 Join Using Whole Tuple Redeclustering (Algorithm B—Clone Join) ............. 43
43 Analytical Modeling: Effect of Join Selectivity ($|R| = |S| = M$, replication probability = 0.07, fertility ratio = 0.2) .......................................................... 49
44 Analytical Modeling: Effect of Replication Probability ($|R| = |S| = M$, join selectivity = 0.5e-6, fertility ratio = 0.2). ..................................................... 50
45 Analytical Modeling: Effect of Fertility Ratio ($|R| = |S| = M$, join selectivity = 0.5e-6, replication probability = 0.07) ..................................................... 50
46 Join Drainage and Roads (Replication Probability = 0.07, Fertility Rate = 0.20, Join Selectivity = 0.53e - 6) ................................................................. 53
Effect of Join Selectivity: Join Drainage and Roads (Replication Probability = 0.07, Fertility Rate = 0.20, \textbf{Join Selectivity} = 0.27e – 6) .......................................................... 53

Effect of Fertility Ratio: Join Drainage and Roads (Replication Probability = 0.07, \textbf{Fertility Rate} = 0.11, \textbf{Join Selectivity} = 0.53e – 6) .......................................................... 54

"Stitched" Data, Effect of Replication: Join Stitched-Road and Stitched-Rails (\textbf{Replication Probability} = 0.18, Fertility Rate = 0.23, \textbf{Join Selectivity} = 1.3e – 6) ................................. 54

One Approximation per Tuple ................................................................. 56

Fragmented Approximations ..................................................................... 56

Parallel Aggregate Evaluation, Naive Approach: Operator Tree .................. 59

Parallel Aggregate Evaluation, Naive Approach: Operator Tree Instance ........ 59

Operator Tree for Parallel Aggregate Evaluation ....................................... 60

Spatial Semi-join .................................................................................. 61

Find the Closest River to Every Large City .............................................. 63
# List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Classification of Various Spatial Join Algorithms</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Wisconsin TIGER Data</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>Sequoia Data.</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Detailed Cost breakdown, TIGER Data. Join Roads with Hydrography (All times are in seconds)</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Detailed Cost breakdown, Clustered Tiger Data. Join Roads with Hydrography (All times are in seconds)</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Detailed Cost breakdown, Sequoia Data. Join Polygons with Islands (All times are in seconds)</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>TIGER Data, Join Road with Hydrography (All times are in seconds)</td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td>Clustered TIGER Data, Join Road with Hydrography (All times are in seconds)</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>Sequoia Data, Join Polygons with Islands (All times are in seconds)</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>DCW Cities and Drainage Data Set</td>
<td>30</td>
</tr>
<tr>
<td>11</td>
<td>System Parameters.</td>
<td>45</td>
</tr>
<tr>
<td>12</td>
<td>Data Characteristics.</td>
<td>45</td>
</tr>
<tr>
<td>13</td>
<td>DCW Data Set.</td>
<td>52</td>
</tr>
<tr>
<td>14</td>
<td>Stitched DCW Data Set.</td>
<td>55</td>
</tr>
<tr>
<td>15</td>
<td>Effect of Multiple Finer Approximations on Join Drainage and Road</td>
<td>56</td>
</tr>
<tr>
<td>16</td>
<td>Effect of Multiple Finer Approximations on Join Stitched Road and Rail</td>
<td>56</td>
</tr>
<tr>
<td>17</td>
<td>DCW Cities and Drainage Data Set</td>
<td>62</td>
</tr>
</tbody>
</table>
# Contents

Abstract i

Acknowledgements ii

1 Introduction 1

1.1 Motivation ................................................................. 1

1.2 Thesis Contents ............................................................. 2

1.3 Thesis Outline ............................................................ 3

2 Spatial Joins 5

2.1 Motivation for Efficient Spatial Join Algorithms ................... 5

2.2 Previous Work ............................................................. 6

2.3 Partition-Based Spatial-Merge Join .................................... 7

2.3.1 Filter Step ............................................................... 8

2.3.2 Refinement Step: Checking the Candidate OID pairs for Exact Match 10

2.3.3 Determining the Number of Partitions ............................ 11

2.3.4 Choosing a Spatial Partitioning Function ............................ 11

2.3.5 Handling Partition Skew .............................................. 13

2.4 Performance Evaluation ................................................ 14

2.4.1 Indexed Nested-Loops Join ............................................ 14

2.4.2 R-tree-Based Join Algorithm ......................................... 14

2.4.3 Methodology ............................................................ 15

2.4.4 Joins When None of the Indices Pre-exist ....................... 16

2.4.5 Joins in the Presence of Pre-existing Indices ...................... 21

2.4.6 CPU Costs ............................................................... 22
2.4.7 Effect of Spatial Partitioning Function Parameters on PBSM ........................................ 24
2.5 Conclusions .................................................................................................................... 25

3 Spatial Aggregates ......................................................................................................... 26
  3.1 What is a Spatial Aggregate? ....................................................................................... 26
  3.2 Previous Work ............................................................................................................. 26
  3.3 Nearest Neighbor Search Using an Index Independent Algorithm ......................... 27
  3.4 Index Independent Algorithm For General Spatial Aggregation ......................... 28
  3.5 Spatial Aggregation with Joins .................................................................................. 28
  3.6 Performance Evaluation ............................................................................................. 29
  3.7 Conclusions ................................................................................................................ 31

4 Declustering Spatial Data ............................................................................................... 33
  4.1 Spatial Data and Relational Parallelism ...................................................................... 33
  4.2 Related Work ............................................................................................................... 34
  4.3 Spatial Declustering Techniques ................................................................................. 35
    4.3.1 Declustering Spatial Data Using Replication ...................................................... 35
    4.3.2 Partial Spatial Surrogates ..................................................................................... 37
  4.4 Conclusions ................................................................................................................ 39

5 Parallel Spatial Join Algorithms ...................................................................................... 40
  5.1 Execution Phases of Parallel Spatial Joins ................................................................. 40
  5.2 Design Space ............................................................................................................... 41
  5.3 Analytical Model ......................................................................................................... 44
    5.3.1 Notations ............................................................................................................. 44
    5.3.2 Cost Formula for the Clone Join ......................................................................... 45
    5.3.3 Cost Formula for the Shadow Join ..................................................................... 47
  5.4 Analytical Predictions ................................................................................................. 48
    5.4.1 Effect of Join Selectivity ...................................................................................... 49
5.4.2 Effect of Replication Probability ........................................... 49
5.4.3 Effect of Fertility Ratio .................................................. 49
5.4.4 Summary of the Analytical Performance Comparisons ............... 51
5.5 Experimental Performance Comparison ................................... 51
  5.5.1 Data Sets .............................................................. 51
  5.5.2 Testbed Description and DBMS Configuration ........................ 52
  5.5.3 Validation of the Analytical Model ................................... 52
  5.5.4 Experimental Results ............................................... 52
5.6 Conclusions .................................................................... 57

6 Parallel Spatial Aggregates .................................................. 58
  6.1 Parallel Spatial Aggregate Algorithm .................................... 58
    6.1.1 Naive Evaluation .................................................. 58
    6.1.2 Spatial Semi-Join .................................................. 60
  6.2 Performance Evaluation .................................................. 61
  6.3 Conclusions .............................................................. 62

7 Conclusions .................................................................... 64
  7.1 Summary .................................................................. 64
  7.2 Future Work .................................................................. 65

Bibliography ..................................................................... 66
Chapter 1

Introduction

1.1 Motivation

Applications that store and manipulate spatial data have been around for a long time, and the systems that are used to support these applications are constantly evolving. In the very old days, maps were drawn on some physical media such as paper or parchment, and spatial operations such as charting of routes was largely a manual process. Then, in more recent times, many of these processes were automated by first digitizing the spatial information and then querying the digitized data. In the earlier stages of this trend towards digitization, the digitized data was stored in operating system files and tools were built, usually using scripts in some scripting language, to extract information from these files. However, as the volume of the data that was used by these applications increased, storing data in operating system files became cumbersome. File systems provide limited organizational facilities (usually only directories), and require the user to name every file and organize them into meaningful hierarchies. With large data sets, this management becomes very cumbersome. Subsequently, to ease the data management problem, database systems were employed to store the data. Using a database system also allowed the applications to exploit the declarative querying facilities provided by the database system. These database systems were commercially available database systems that use a model called the relational model. In this model, information is organized into tables. A table is a set of tuples of the same type. A tuple is a list of typed attributes. The type of an attribute is limited to a set of predefined system types (that usually does not include spatial types). Using a relational database system did ease some of the data management problem, but they were still awkward to use and not very efficient for spatial applications.

One limitation of both the relational database approach and the file system approach is that neither of these solutions fundamentally understand spatial data types. In both these solutions, spatial data is just stored in the system as an uninterpreted array of bytes. The application program has to interpret these raw bytes as spatial data and perform operations on the spatial data. Consequently, the declarative querying facilities of the database are used only for querying the non-spatial portion of the data, with every application building its own spatial querying features. Clearly, a more integrated solution was needed.

Similar problems also arose in other application domains (like multimedia, time-series, CAD) where a relational database system was used as a repository of the data, but not for actual manipulation of the data. Database models began evolving to solve these problems, and have lead to the currently prevalent paradigm that allows “objects” to be stored inside a relational model. Attributes of tuples in traditional relational systems are limited to simple types like numeric, dates and character strings, but the new model, called the object-relational model, allows the extension of these basic types to include types like images, videos, points, polygons, circles, polylines,
etc [SK91, DKL+94]. The object–relational model also allows functions to be defined on these types, and allows these functions to appear in queries that are posed to the system. Such a database system can both store and manipulate complex types inside the database system, making them easier to use. However, to make these database systems efficient, special techniques are required. This thesis explores the techniques that a database system can use to efficiently support spatial applications. More specifically, this thesis explores efficient algorithms for evaluation spatial operations.

1.2 Thesis Contents

This thesis has two main parts. The first part explores efficient techniques for evaluating two important spatial operations: the spatial join and the spatial aggregate. A spatial join combines two spatial data sets based on some spatial relationship between the elements of the two data sets. For example, given a set of landuse polygons and a set of soil erosion polygons, a spatial join can overlap these two sets of polygons to find all landuse polygons that are undergoing soil erosion. A spatial aggregate summarizes the input by applying a spatial aggregate function. An example of spatial aggregation is locating the closest toxic waste dump to a given city.

The second part of the thesis explores the use of parallelism to handle large volumes of spatial data. Parallel database systems are popular in the relational world for handling large volumes of business data. Spatial data sets are also getting larger and larger, and it is natural to apply parallelism to handle these large data sets. The second part of the thesis first explores alternative ways of storing spatial data in a parallel database system, and then explores parallel algorithms for spatial join and spatial aggregate operations.

![Figure 1: Evolution of Database Systems](image-url)

Seen in the context of the evolutionary path of database systems (see Figure 1), the first part of the thesis focuses on the techniques that can be used to support spatial applications in a database system that falls in the category labeled B in the Figure, while the second part of the thesis deals with techniques for supporting spatial applications in a database system that falls in the category labeled D. The techniques that are considered in this
thesis are efficient algorithms for evaluating the spatial join and the spatial aggregate operations, and strategies for distributing spatial data across nodes in a parallel database system. Spatial indexing techniques, data model issues, and query language issues are not part of this thesis.

1.3 Thesis Outline

The outline of the rest of the thesis is as follows. Chapter 2 presents a new (centralized) algorithm for performing the spatial join and compares its performance with other centralized spatial join algorithms. Chapter 3 presents an algorithm for evaluating a spatial aggregate using a spatial index to speed up the evaluation.

The next three chapters explore the use of parallelism in a spatial database system. Chapter 4 presents and evaluates various strategies for declustering spatial data. Chapter 5 explores the design space of various parallel spatial join algorithms, and also contains performance results from implementing these algorithms. Chapter 6 presents a new algorithm for evaluating the spatial aggregate in a parallel environment. Finally, the conclusions of this thesis are contained in Chapter 7.
PART I: Centralized Spatial Database Techniques
Chapter 2

Spatial Joins

This chapter discusses algorithms for evaluating the spatial join operation in a centralized spatial database system. A classification of various spatial join algorithms is presented, followed by the description of a new spatial join algorithm called the Partition Based Spatial-Merge Join (PBSM). A performance comparison of this algorithm with other spatial join algorithms demonstrates the feasibility, and applicability of the PBSM join algorithm.

2.1 Motivation for Efficient Spatial Join Algorithms

A spatial database system has to store spatial data types with simple geometries such as points, lines, polygons, and surfaces, and more complex geometries such as swiss-cheese-polygons (which are polygons with holes) that are derived from the simpler geometric types. Besides storing various types of spatial data, a spatial database system must also efficiently support queries on the spatial data. Spatial database users frequently need to combine two spatial inputs based on some spatial relationship between the objects in the two inputs. For example, map overlap, which requires combining two maps to produce a third, is an important operation in a spatial database [Bur86, MGR91]. This operation of combining two inputs based on an element-wise spatial relationship is called a spatial join. Spatial joins, just like their counterparts in a relational system, are an expensive operation. Consequently, efficient spatial join algorithms are a critical component of any spatial database system.

As an example of a spatial join, consider a database that stores the following information:

**FaultLines** (category: **String**, shape: **PolyLine**)

**Rivers** (name: **String**, shape: **PolyLine**)

**Cities** (name: **String**, population: **Integer**, boundary: **Polygon**)

A query to find all rivers that lie on fault lines is a spatial join where the join condition is the intersection between Rivers.shape and FaultLine.shape. Another example of a spatial join would be a query that finds all cities that lie within 50 miles of any fault line.
2.2 Previous Work

Since the representation of a spatial object can be very large (for example, a spatial object representing a city boundary might require thousands of points to represent the exact geometric shape), spatial operations, including the spatial join, typically operate in two steps [Ore90]:

• **Filter Step:** In this step, an approximation of each spatial object, such as its minimum bounding rectangle, is used to perform an element-wise pairing between the tuples from the two input data sets. This step produces *candidates* that are a superset of the actual result. These candidates are usually represented as a pair of object identifiers.

• **Refinement Step:** In this step, each candidate is examined (which usually requires fetching a pair of objects from disk) to check if it is part of the result. This check generally requires running a CPU-intensive computational geometry algorithm.

Most of the spatial join algorithms that have been proposed in the literature only evaluate the filter step of the join. Relatively little attention has been given to the refinement step. Various techniques for improvement of the refinement step are proposed and evaluated by Brinkhoff, Kriegel, Schneider and Seeger in [BKSS94]. In this section, we classify the various algorithms for evaluating the filter step of the spatial join. For convenience, throughout this section the term spatial join is used to refer to the filter step of the spatial join.

One of the earliest solutions to the spatial join is based on using approximate geometry, wherein the universe of the spatial data is regularly decomposed by superimposing a grid on it [Ore86, OM88]. Each element of the grid is called a pixel, and spatial objects are approximated by pixels that overlap them. Each pixel, which is described by its spatial location, is transformed into a 1-dimensional domain by applying a mapping called the z-order. The transformed values, which are called z-values, are then used in a spatial join algorithm that merges two sequences of z-values. The z-values, being 1-dimensional values, can be stored in a traditional indexing structure such as a B-tree [OM84]. The performance of the spatial join algorithm using z-values was found to be sensitive to the choice of the grid [Ore89]. Choosing a fine grid increases the efficiency of the filtering technique, but it also increases the space requirement, since a larger number of z-values are required to approximate an object.

In the relational domain, Valduriez proposed the use of join indices [Val87] to improve the performance of the relational join operator. Drawing an analogy from the join indices, Rotem [Rot91] proposed a spatial join index that partially precomputes the results of a spatial join. The algorithm for building the spatial join index requires grid files for indexing the spatial data, and uses these grid files to compute the spatial join index. Grid files [NHS84] and k-d trees [Ben75, Ben79] have also been employed for evaluating multi-attribute joins in the relational domain [KHT89, HNKT90, BHF93]. These methods can also be used for evaluating the filter step by storing the bounding box of the spatial objects as points in a higher dimension [BHF93].

Recently, spatial index structures such as R-trees [Gut84], R+-trees [CFR87], R*-trees [BKSS90], and PMR quad trees [NS86] have been used to speed up the evaluation of the spatial join. Using analytical models, Günther compares join algorithms that use generalization trees (which is a class of tree structures that includes the R-tree, R*-tree and R+-tree) with the nested-loops join and join indices [Gün93]. This study concludes that for low join selectivities, join indices usually provide the best join performance, but for higher join selectivities generalization
trees are more efficient. The proposed join algorithm, using the generalization trees, is similar to the join algorithm on R-trees proposed by Brinkhoff, Kriegel and Seeger [BKS93]. This algorithm can be used only if an R-tree index exists on both the join inputs, and can be described as a synchronized depth-first search of both indices, with the two depth-first searches being guided by hints from each other. This algorithm was modified by Huang, Jing and Rundensteiner [HJR97] to achieve better memory utilization by changing the depth-first traversal to a breadth-first traversal of the two trees. Similar tree joins have been proposed for other data structures. Hoel and Sanet [HS95] propose a tree join algorithm for the PMR quad tree, and compare the efficiency of variants of the PMR quad tree with variants of the R-tree [HS95].

When one of the inputs to the spatial join does not have a spatial index, Lo and Ravishankar [LR94] propose building a seeded tree index on that input. A seeded tree is a R-tree that is allowed to be height unbalanced. The algorithm for constructing the seeded tree uses the existing index on one of the two inputs as a starting point, and tries to minimize the number of random I/Os incurred during the tree construction. The two indices are then joined using the tree join algorithm described by Brinkhoff, Kriegel and Seeger [BKS93]. Lo and Ravishankar [LR95] have extended this work to handle the case when neither of the inputs have an index. In their approach, spatial sampling techniques are used for constructing seeded trees on both inputs, and the seeded trees are joined using the tree join algorithm of [BKS93].

Similar to the seeded tree approach, Koudas and Sevcik [KS97] propose building a filter tree on the inputs and joining the two filter trees. A filter tree, like a Quad Tree, is based on a regular decomposition of space. However, unlike other spatial indices, a filter tree can have objects at the non-leaf nodes. The join algorithm first constructs the two filter trees and joins the two filter trees using an n-way merge algorithm. The merge algorithm closely resembles the z-values merge algorithm of Orenstein [Ore89].

The problem of finding pairwise intersection between two sets of rectangles has been extensively studied in the VLSI domain [MC80], and numerous solutions exist for the case when both the input set of rectangles fit in memory [PS88]. In [GS87], Gütting and Shilling examine the rectangle intersection problem when the inputs are too large to fit in memory, and analyze the time and space complexity of two algorithms that are based on external computational geometry algorithms. However, these algorithms are not very efficient with respect to the number of disk I/Os, and in some cases require logarithmic number of passes over the input.

Concurrent with our work on PBSM, Lo and Ravishankar have proposed a spatial hash join algorithm [LR96] that is, in many aspects, similar to PBSM. The spatial hash algorithm first partitions both the inputs, and then joins each of the partitions. Upper levels of a seeded tree are used for the partition function, and a filtering technique is employed during the partitioning phase. A performance study, based on counting the number of disk I/Os, is also presented by Lo and Ravishankar [LR96], but that study ignores the very expensive refinement step.

To summarize, we can classify all these algorithms as shown in Table 1.

2.3 Partition-Based Spatial-Merge Join

This section describes a new algorithm, called the Partition-Based Spatial-Merge (PBSM) join, for evaluating a spatial join. For the sake of concreteness, let R and S denote the two input data sets (relations in a relational
<table>
<thead>
<tr>
<th>Transform the approximation into another dimension</th>
<th>Require Use of an Index</th>
<th>Operate without an Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Z-values [OM84]</td>
<td>• Z-values [Ore86, OM88]</td>
<td></td>
</tr>
<tr>
<td>• Join Indices [Rot91]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Grid Files [HNKT90, BHF93]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• kd-trees [KHT89, HNKT90]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Classification of Various Spatial Join Algorithms

database system) to the join. We assume that the data sets are a sequence of tuples, and that each tuple has a spatial attribute that is used in the join condition. We also assume that the system has a unique identifier for each tuple. This unique identifier is referred to as the object identifier (OID) of the tuple.

The PBSM algorithm operates in the following two steps.

• **Filter Step:** The spatial attribute involved in the join may be a complex spatial feature such as a polygon, a polyline, or a swiss-cheese polygon. In this step, the PBSM algorithm makes use of an approximation of the spatial feature to get a “rough estimate” of the characteristics of the spatial attribute. The minimum bounding rectangle (MBR), is used as an approximation in this step. The filter step uses partitioning to partition large input data sets into smaller chunks. A computational geometry plane-sweeping technique is used to join the chunks. The result of the filter step is a set of OID pairs such that one OID of the pair refers to a tuple from the input data set R and the other OID refers to a tuple from the input data set S. Furthermore, for each pair, the MBR of the spatial join attribute of the R tuple overlaps with the MBR of the spatial join attribute of the S tuple.

• **Refinement step:** Since two non-overlapping spatial features can have overlapping MBRs, and since the filter step “joins” the input data sets based on the MBR of the joining attributes, the filter step generally will produce a proper superset of the join result. In the refinement step, the R and S tuples represented by the OID pair produced by the previous step are fetched from disk, and their join attributes are examined to determine if the join predicate is actually satisfied.

The next section describes the filter step in detail, and the section following that describes the refinement step.

### 2.3.1 Filter Step

The filter step of the PBSM algorithm begins by reading tuples from the data set R. For each tuple of the data set R, the MBR of the joining attribute and the OID of the tuple, which is collectively called a key-pointer element, is appended to a temporary relation on disk. Let us denote this relation by $R^{kp}$. Similarly, the data set S is read and a relation $S^{kp}$ is produced. The goal of the filter step is to “pair” tuples from R and S such that the MBRs of their join attributes overlap. $R^{kp}$ and $S^{kp}$ have the MBRs for the join attributes of both the inputs R and S. The problem then simplifies to finding all MBRs in $R^{kp}$ that intersect with some MBR in $S^{kp}$. Rectangle
intersection (the MBRs are rectangles) has been extensively studied in the computational geometry field [PS88].

Given two sets of rectangles, such that both the sets fit entirely in main memory, efficient computational geometry algorithms, based on plane-sweeping techniques, exist for reporting all pairs of intersecting rectangles between the two sets. Now, if both \( R^{kp} \) and \( S^{kp} \) fit in memory, then a plane-sweeping algorithm can be used to find all pairs of \( R^{kp} \) and \( S^{kp} \) key-pointer elements that have overlapping MBRs. For such “matching” key-pointer elements pairs, the OID information is extracted, and the OID pair is added to the output of this step.

If \( R^{kp} \) and \( S^{kp} \) are too large to fit entirely in memory, then each is divided into \( P \) (non-disjoint) partitions \( R_i^{kp}, R_2^{kp}, \ldots, R_P^{kp} \) and \( S_1^{kp}, S_2^{kp}, \ldots, S_P^{kp} \) respectively. These partitions are formed in a way such that for each key-pointer element in a partition \( R_i^{kp} \), all the key-pointer elements of \( S^{kp} \) that have an overlapping MBR are present in the corresponding \( S_i^{kp} \) partition. Furthermore, the size of the partitions are such that for each \( i \) (1 \( <= i \leq P \) ) \( R_i^{kp} \) and \( S_i^{kp} \) can both fit simultaneously in memory.

To form these partitions, a spatial partitioning function is used. The spatial partitioning function works as follows:

- From the catalog information for the joining attribute of input \( R \), the algorithm estimates the universe of the input. The universe for a particular spatial join attribute is the rectangle that is the minimum cover of the join attribute of all the tuples in the input.
- The universe is then decomposed into a \( P \) subparts (\( P \) is the number of partitions). As an example, consider Figure 2 where the number of partitions is 4.
- Finally, the spatial partitioning function (see Section 2.3.4 for more details) is applied to the MBR of a key-pointer element. The partitioning function determines all the subparts of the universe with which the MBR overlaps, and inserts the key-pointer element into each partition corresponding to these subparts. Thus, if a MBR overlaps with multiple subparts of the universe, then it is inserted into multiple partitions. For example, the key-pointer element for the object shown in Figure 2, will be inserted into partitions 0 and 2.

![Figure 2: The Spatial Partitioning Function.](image-url)
MBR which has a smaller MBR.xl value is selected. Let \( r \) denote this MBR, and let us assume that \( r \) belongs to the input \( R_i^{kp} \). Using the \( r.xu \) value, the key-pointer elements of the input \( S_i^{kp} \) are scanned until a key-pointer element whose MBR has a MBR.xl value greater than \( r.xu \) is reached. Effectively, all the elements of \( S_i^{kp} \) that overlap with \( r \) along the x-axis are scanned. Each of these elements of \( S_i^{kp} \) is then checked for overlap with \( r \) along the y-axis\(^1\). If an overlap exists, then the OID pair corresponding to the OIDs in the key-pointer elements is added to the result (the result of the filter step is a set of OID pairs). After this, \( r \) is marked as processed and is removed from consideration for the input \( R_i^{kp} \). The algorithm continues by picking from the unprocessed part of the inputs \( R_i^{kp} \) and \( S_i^{kp} \) the element that has the smallest MBR.xl value. The smallest element is then “joined” with elements in the other input. This process continues until one of the two inputs has been fully processed.

As an example consider Figure 3, where the input \( R \) has 2 elements, and the input \( S \) has 3 elements. The algorithm starts with \( R1 \) (since it has the smallest MBR.xl) and scans the elements in \( S \) until it reaches \( S2 \). Along the way, \( S0 \) and \( S1 \) are checked for intersection with \( R1 \) along the y-axis. \( S1 \) checks positively for intersection with \( R1 \), and the OID pair \( <OID_{R1}, OID_{S1}> \) is added to the intermediate result. \( R1 \) is then marked as processed, and the algorithm proceeds with \( S0 \), then \( S1 \), and finally \( R2 \).

![Figure 3: Spatial Sort Merge Example.](image)

2.3.2 Refinement Step: Checking the Candidate OID pairs for Exact Match

After joining each pair of partitions, the result is a relation whose tuples have the form \( <OID_R, OID_S> \), such that the MBR of the joining attribute of the \( R \) tuple corresponding to \( OID_R \) overlaps with the MBR of the joining attribute of the \( S \) tuple corresponding to \( OID_S \). Since the partitioning in the filter step might insert a tuple into multiple partitions, there could be duplicates in this relation. The refinement step eliminates these duplicates, and examines the actual \( R \) and \( S \) tuples to determine if the attributes actually satisfy the join condition.

To avoid random seeks in fetching the \( R \) and \( S \) tuples, a strategy similar to that used by Valduriez [Val87] is employed. First, the OID pairs are sorted using \( OID_R \) as the primary sort key and \( OID_S \) as the secondary sort key. Duplicates entries are eliminated during this sort. Next, as many \( R \) tuples as can fit in memory are read from disk along with the corresponding array of \( <OID_R, OID_S> \) pairs. The \( OID_R \) part of this array is “swizzled” to point to the \( R \) tuples in memory, and then the array is sorted on \( OID_S \) (this makes the accesses to \( S \) sequential). The \( S \) tuples are then read sequentially into memory, and the join attributes of the \( R \) and the \( S \) tuple are checked to determine if the join condition is satisfied.

\(^1\)This check for overlap can be speeded up by organizing the MBRs of \( S_i^{kp} \) that overlap with \( r \) along the x-axis in an interval-tree [FS88].
2.3.3 Determining the Number of Partitions

The number of partitions for the PBSM algorithm can be estimated as follow. Let $\|R\|$ represent the cardinality of the input $R$, and $\|S\|$ represent the cardinality of the input $S$. Also, let $M$ represent the size of the main memory in bytes, and let $Size_{key-\text{ptr}}$ denote the size of a key-pointer element (which is a $\langle MBR, OID \rangle$ pair) in bytes. Since, the plane-sweep algorithm used in merging the partitions requires both the partitions, $R^k_t$ and $S^k_t$, to fit entirely in memory, the number of partition $P$ is computed as:

$$P = \lceil \frac{(\|R\| + \|S\|) \times Size_{key-\text{ptr}}}{M} \rceil$$  

(1)

2.3.4 Choosing a Spatial Partitioning Function

We now explore some of the alternatives that exist in selecting a spatial partitioning function. The spatial partitioning function described in Section 2.3.1 decomposes the universe into $P$ subparts (where $P$ is the number of partitions determined by Equation 1). However, in the presence of a non-uniform distribution, this partitioning function could produce partitions that have very different sizes. As an example, consider Figure 4 where the number of partitions is 4. Here, most of the tuples are in the top left corner, and the spatial partitioning function will map all these tuples to partition 0. Partitions 2 and 3, on the other hand will have very few tuples.

![Figure 4: Spatial Partitioning Skew](image-url)

To remedy this situation, the partitioning function used in PBSM decomposes the universe regularly into $NT$ tiles, where $NT$ is greater than or equal to $P$. Starting from the upper left corner and going across, the tiles are numbered from 0 to $NT - 1$. Each tile is mapped to a partition using a scheme such as round robin or hashing. For example, consider Figure 5, where the universe is divided into 12 tiles, the number of partitions is 3, and tiles are mapped to partitions using a round robin scheme. Thus tiles 0, 3, 6, and 9 are mapped to partition 0, tiles 1, 4, 7 and 10 are mapped to partition 1, and tiles 2, 5, 8 and 11 are mapped to partition 2. To apply the spatial partitioning function to a MBR, all the tiles that overlap with the MBR are determined, and, for each tile, the key-pointer element corresponding to the MBR is inserted into the corresponding partition. Thus, if a MBR overlaps with tiles from multiple partitions, then its key-pointer element will be inserted into all those partitions. Consequently, the key-pointer element for the object shown in Figure 5, will be inserted into partitions 0, 1 and 2.

The spatial partitioning function just described is the spatial analog of virtual processor round robin partitioning for handling skews in parallel relational joins [DNS92]. A similar partitioning function has been independently proposed for redundancy-based declustering of spatial objects in a parallel spatial database [TY95], but in that proposal the number of tiles always equals the number of partitions.
The design space for choosing the spatial partitioning function has two axes: the number of tiles used in the partitioning function, and the tile-to-partition mapping scheme. Decomposing the universe into small tiles produces many small containers, which are easier to pack into partitions to produce a more uniform partition distribution. However, spatial objects that span tiles from multiple partitions must be replicated in each partition, thereby increasing the replication overhead. For the tile-to-partition mapping scheme, one could use either round robin or hashing on the tile number.

To explore these alternatives, we have chosen two data sets. The first data set is derived from the TIGER/Line files [Tig], and represents roads in the state of Wisconsin. This data set is 62.4MB in size, and contains 456,613 tuples. The second data set contains the polygon data from the Sequoia benchmark [SFGM93]. This data set contains 58,115 polygons and is 21.9MB in size.

First, we explore the design space of the spatial partitioning function using the Tiger data. Figure 6 shows the effect of increasing the number of tiles, and choosing different tile-to-partition mapping schemes. The graph uses the coefficient of variation\(^2\) of the distribution of the tuples in each partition as its metric. A perfect spatial partitioning function would be one that assigns an equal number of tuples to each partition, and, consequently, would have a coefficient of variation of 0 for the distribution of the tuples in each partition. From Figure 6, the following observations can be made. First, using a large number of tiles and hashing on the tile number gives a good partitioning function. Second, all the partitioning functions improve as the number of tiles is increased. This is because with more tiles, “dense” regions get subdivided into more tiles, and these tiles can be mapped to different partitions. Third, for a given number of tiles, the partitioning function yields a more uniform distribution for a smaller number of partitions (compare the graphs for hashing with 4 and 16 partitions). This behavior, once again, is because the distribution of tiles that cover “dense” regions is better with a smaller number of partitions. For example, when the universe is decomposed into 25 tiles, 13 tiles cover 81.5% of the data. These 13 tiles can be spread across 4 partitions more uniformly than over 16 partitions, and, as a result, the coefficient of variation for 4 partitions is lower than that for 16 partitions.

Figure 7 measures the replication overhead—the increase in the number of tuples created due to replication during partitioning—for various number of tiles. The figure shows that, for the Tiger data set, the replication overhead is very modest even for a very large number of tiles (increasing the number of tuples by 4.8% for 4000 tiles in a 16 partition configuration). The figure also shows some spikes in the curve for round robin. With round robin, when the number of tiles is an integral multiple of the number of partitions, it is possible for an entire column

\(^2\)The coefficient of variation is defined as the standard deviation divided by the mean.
to get mapped to a single partition. This mapping is equivalent to having fewer tiles (each column now behaves like a single tile), and, consequently, fewer tuples have to be replicated. However, at these points, the partitions produced by the partitioning function are less balanced (observe the jumps in Figure 6 for round robin with 16 nodes).

For the Sequoia data, we found that the effect of increasing the number of tiles on the tile-to-partition mapping scheme is very similar to the effect on the Tiger Data. However, the replication overhead, which is shown in Figure 8, is much higher. The Sequoia data set shows a higher overhead because it has very different data characteristics. The Sequoia data set has many polygon features that span a large area of the space. These have a higher chance of getting replicated, causing the replication overhead to be high. On the other hand, the road segments in the Tiger data set are small in terms of their (spatial) length. In the Tiger data set, a single road like a highway is broken down into many smaller road segments. Each road feature has associated information like the zip code of the region through which the segment of the road passes. If a highway cuts across regions with different zip codes, it gets broken into many smaller road segments. Consequently, with the Tiger data set, there are very few road segments that overlap multiple tiles, which in turn causes the replication overhead to be low.

2.3.5 Handling Partition Skew

Similar to the partition skew problem for Grace Join [KTM83, DKO'84, Sha86, KNT89], it is possible for the PBSM algorithm to end up with partition pairs that do not fit entirely in memory (for example, if most of the data is concentrated in a very small cluster). One possible way to handle this problem would be to dynamically repartition the overflow partition pair. Another alternative is to increase the number of partitions (limited to the number of available main memory buffer pages) and using schemes similar to those used by the Adaptive Hash join algorithm [ZG90]. However, the current implementation of PBSM does not incorporate any of these
techniques.

2.4 Performance Evaluation

In this section, we compare the PBSM join algorithm with two other spatial join algorithms. The first algorithm is based on the traditional indexed nested-loops algorithm and the other is based on the R-tree join algorithm [BKS93]. These algorithms use spatial indices, and were chosen because most spatial databases support some form of spatial indexing (for example, R-trees in Illustra [Ube94]). Such systems can easily use these index-based join algorithms. This study is not meant to be a comprehensive performance study of all possible spatial join algorithms (refer to Table 1 for a classification of spatial join algorithms). However, the algorithms that we study are alternatives that can be used in a spatial database system that does not transform approximations of spatial objects into another domain (e.g., a 1-dimensional domain). To the best of our knowledge, most commercial spatial database systems do not transform the approximations of spatial objects into another domain (for example, ARC/INFO [Arc95], and Illustra [Ube94]).

The remainder of this section is organized as follows. First the index nested-loops and the R-tree-based join algorithms are described. Then, the methodology used in the study is described, and, finally, the results of the study are presented.

2.4.1 Indexed Nested-Loops Join

Let \( R \) and \( S \) denote the two relations being joined, and assume that \( R \) has fewer tuples than \( S \). If neither join input has an index on the joining attribute, the indexed nested-loops join algorithm first builds an index on the smaller input \( R \). The index is built using a bulk loading mechanism that reads the extent \( R \) and extracts the key-pointer information \((<MBR, OID>)\) for each tuple. The key-pointer information is then spatially sorted based on the \( MBR \). Spatial sorting is accomplished by transforming the center point of the \( MBR \) into a one dimensional Hilbert value ([Hi91] as referred to in [MJFS95]), and using this value for ordering the key-pointer information. This sorting step brings together key-pointer pairs whose joining attributes are spatially close. The spatial index, which in our case is a R*-tree, is then built in a bottom up fashion [DKL+94]. After building the index on the join attribute of \( R \), a scan is started on \( S \). Each tuple of \( S \) is used to probe the index on \( R \). The result of the probe is a set of (possibly empty) \( OIDs \) of \( R \). The tuples of \( R \) corresponding to these \( OIDs \) are then fetched (from disk, if necessary) and checked with the \( S \) tuple to determine if the join condition is satisfied. Fetching each \( R \) tuple from disk will generally incur a random disk I/O.

2.4.2 R-tree-Based Join Algorithm

For this algorithm, we first bulk load an R*-tree index on the joining attribute of the two input relations. The two indices are then joined using the R-tree join algorithm proposed by Brinkhoff, Kriegel and Seeger [BKS93]. The R-tree join algorithm performs a synchronous depth-first traversal of the two trees. The traversal starts with the roots of the two R-trees, and moves down the levels of the two trees in tandem until the leaf nodes are reached.
At each step, two nodes, one from each tree, are joined. Joining two nodes requires finding all bounding boxes in the first node that intersect with some bounding box in the other node. The child pointers corresponding to such matching bounding boxes are then traversed (resulting in a depth-first traversal).

The R-tree join algorithm proposed by Brinkhoff, Kriegel and Seeger [BKSS03] only performs the filter step of the spatial join, and produces a set of candidate OID pairs corresponding to the objects whose MBRs intersect. The objects corresponding to these OIDs then have to be fetched and checked to determine if the join predicate is actually satisfied. For this step, we use the same refinement strategy that was used by the PBSM join algorithm (refer to Section 2.3.2).

### 2.4.3 Methodology

For the performance comparison, we implemented each of these algorithms, namely, indexed nested-loops join, R-tree-based join and the PBSM join in Paradise [DKL+94]. Paradise is a database system that handles geospatial types of applications. Paradise supports storing, browsing, and querying of geographic data sets. It uses an extended-relational data model and supports an extension of SQL as its query language. Paradise uses SHORE [CDF+94] as its storage manager for persistent objects.

The machine used for the study was a Sun SPARC-10/51 with 64 MBytes of memory, running SunOS Release 4.1.3. One Seagate 2GByte disk (3.5” SCSI, model # ST 12400N) was used as a raw device to hold the database. The log for the system was kept on a second 2GByte Seagate disk.

For the performance study, the PBSM algorithm used 1024 tiles for the spatial partitioning function. We explored the effect of the number of tiles on the execution time of PBSM (refer to Section 2.4.7), but found that changing the number of tiles had a very small effect on the overall execution time (less than 5%).

The performance study was carried out in two parts. The first part examined the performance of the three algorithms when neither join input had a pre-existing index, and the second part examined the performance of these algorithms when indices existed on one or both join inputs.

Both parts of the study used three collections of data sets. The first collection was derived from the TIGER/Line files [Tig] for the State of Wisconsin. The TIGER data is developed and distributed by the U.S. Bureau of the Census, and contains detailed geographic and cartographic information for the United States. From the TIGER files, three data sets were extracted (see Table 2). The first data set, called Road, represents the roads, the second data set, called Hydrography, represents basic hydrography features which includes rivers, canals, streams, etc., and the third data set, Rail, represents the railroads. Besides containing a polyline attribute that describes the spatial feature, each tuple in this collection also contains attributes that describes the name, the classification, and the address ranges of the spatial feature. The average number of points in the spatial feature of the Road, Hydrography, and the Rail tuple is 8, 19 and 7 respectively. Two queries were run against this collection. The first query joined the Road data set with the Hydrography data, producing as its result all the intersecting Road and Hydrography features. The result relation consists of 34,166 tuples (about 13.1MB). The second query performed a join between the Road and the Rail data, and produced a 1.4MB result relation that had 4,678 tuples. This query was used to examine the performance of the algorithms when the size of the input data sets was large.
relations differ significantly.

<table>
<thead>
<tr>
<th>Data Type</th>
<th># of Objects</th>
<th>Total Size</th>
<th>R*-tree Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Road</td>
<td>456,613</td>
<td>62.4 MB</td>
<td>24.0 MB</td>
</tr>
<tr>
<td>Hydrography</td>
<td>122,149</td>
<td>25.2 MB</td>
<td>6.5 MB</td>
</tr>
<tr>
<td>Rail</td>
<td>16,844</td>
<td>2.4 MB</td>
<td>1.0 MB</td>
</tr>
</tbody>
</table>

Table 2: Wisconsin TIGER Data

To study the effect of clustering on the join inputs, the second collection was formed by spatially sorting the objects in the first collection.

For the third collection, the islands and polygon data sets from the Sequoia 2000 Storage Benchmark [SFGM93] were used. The polygon data set represents regions of homogeneous landuse characteristics in the State of California and Nevada, while the island data set represents holes in the polygon data (example, a lake in a park). The average number of points in a polygon tuple is 46, and the average number of points in an islands tuple is 35. The query used in this experiment joined the polygons and islands, producing, as a result, those islands that are contained in one or more of the polygons. This result contained 25,260 tuples and was 30.8MB in size. The characteristics of this data are shown in Table 3.

<table>
<thead>
<tr>
<th>Data Type</th>
<th># of Objects</th>
<th>Total Size</th>
<th>R*-tree Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygons</td>
<td>58,115</td>
<td>21.9 MB</td>
<td>3.0 MB</td>
</tr>
<tr>
<td>Islands</td>
<td>21,007</td>
<td>6.2 MB</td>
<td>1.1 MB</td>
</tr>
</tbody>
</table>

Table 3: Sequoia Data.

### 2.4.4 Joins When None of the Indices Pre-exist

This section presents the results of the performance evaluation when neither join input has a pre-existing index. In this case, the Indexed Nested-Loops algorithm builds an index on the smaller inputs and probes it, whereas the R-tree-based algorithm builds both the indices and joins them using the tree join algorithm.

**Comparison Using the TIGER Data**

In the first experiment, the Road data set was joined with the Hydrography data set. Figure 9 shows the execution times of the three spatial join algorithms as a function of the buffer pool size. For this query, the PBSM algorithm is 80-100% faster than the R-tree-based join algorithm, and 93-400% faster than the Indexed Nested-Loops algorithm. The Indexed Nested-Loops builds an index on Hydrography and probes it with the tuples of the Road data set. After probing the index with a tuple, the Indexed Nested-Loops Join fetches the matching Hydrography tuple, and examines it to determine if the join predicate is actually satisfied. For smaller buffer pool sizes, fetching the matching Hydrography tuple generally requires a disk I/O. However, as the size of the buffer pool increases, larger portions of the Hydrography data reside in the buffer pool, and, as a result, the performance of the Indexed Nested Loops Join improves significantly.
Next, the performance of the algorithms was compared by joining the Road data with the Rail data (Figure 10). Since the size of the Rail data is only 2.4MB, and the index on it is only 1.0MB, the index and the data usually fit in the buffer pool (the Rail data pages compete with the Road data pages for buffer pool frames). As a result, the Indexed Nested-Loops performs better than the R-tree-based join algorithm. The cost of the R-tree-based join algorithm is dominated by the cost of building the index on the larger Road data, and the algorithm spends about 85% of its total time building this index.

**Effect of Clustering**

Next, to investigate the effect of clustering, both these queries (joining Road with Hydrography and joining Road with Rail) were run using the clustered Tiger data. The results of this experiment appear in Figures 11 and 12. While joining Roads with Hydrography (Figure 11) PBSM is about 40% faster than the R-tree-based join algorithm, and 60–80% faster than Indexed Nested Loops. While joining Roads with Rail (Figure 12) PBSM is 54–77% faster than the R-tree-based join algorithm, and 42–71% faster than Indexed Nested Loops.

By comparing Figures 9 and 11, and Figures 10 and 12, one can observe that if the inputs to the join are clustered, the performance of all the join algorithms improve. To understand this behavior, consider Figures 13 through 15, which contain detailed cost breakdowns for the query that joins Roads with Hydrography for both the clustered and unclustered cases. For the R-tree-based join algorithm, the total join cost includes the cost of building both indices, the cost of joining the indices, and the cost of fetching the Road and Hydrography tuples from the disk and examining them to determine if the join predicate is actually satisfied (the refinement step). For the Indexed Nested-Loops join, the total cost consists of building the index on Hydrography, and then probing it repeatedly with tuples from the Road data set. For the PBSM algorithm, the total join cost includes the cost of forming the
two partitions, the cost of merging the partitions, and the cost of the refinement step.

First, consider the individual costs of the R-tree-based join algorithm (Figure 13).

• **Index Building Costs:** The indices for this join are built using bulk loading. Refer to Section 2.4.1 for a more detailed description of bulk loading. Bulk loading an index has three costs: the cost of extracting the key-pointers from the input, sorting the key-pointers, and building the index using the sorted key-pointers. When an input is clustered, sorting the key-pointers can be avoided, thereby, reducing the cost of building the index.

• **Tree Joining Cost:** Since bulk loading sorts the keys in the non-clustered scenario, the trees that are built in both the clustered and the non-clustered scenarios are exactly the same. Consequently, the algorithm for joining the two trees performs exactly the same steps in both the cases, and, as a result, clustering has no effect on the time for joining the indices.

• **Refinement Step Cost:** In the refinement step, tuples of R are scanned once, and the tuples of S are scanned multiple times. The refinement step (refer to Section 2.3.2) reads a bunch of R tuples that are physically clustered on the disk, and then reads the S tuples that “spatially match” these R tuples. When the physical order of the tuples on disk is the same as the spatial order, the fetches to the S tuples scan only a small portion of the relation. Consequently, the refinement costs improve with spatial clustering.

Thus, mainly due to large reduction in the index building costs, the R-tree-based join benefits significantly from having the input relations clustered on the join attribute.

Now, consider the Indexed Nested-Loops Join (Figure 14). Clustering has a similar effect on the index building
cost, as sorting the keys can be avoided. Further, for small buffer pool sizes the index on Hydrography cannot fit entirely in memory, and the index probe cost is significantly reduced when the data is clustered. This effect is similar to the behavior of the indexed nested-loops join in the relational domain where sorting the relation that is used to probe the index improves the performance of the join.

Besides the reduction in the refinement cost, clustering reduces the partitioning costs of the PBSM algorithm (Figure 15). The difference in the partitioning costs is more pronounced for smaller buffer pool sizes. This behavior is due to the way partitions are written out to disk. The PBSM algorithm does not manage any of the partition buffers; it simply writes tuples to appropriate partition files, and relies on the SHORE storage manager to flush pages of the partition files to disk. The spatial partitioning function decomposes the universe into tiles and maps the tiles to partitions. When the input is clustered, consecutive tuples are more likely to be in the same tile, and, as a result, get mapped to the same partition. Consequently, this phase incurs at most one disk seek for each tile. When the data is not clustered, the partitions fill up in a random order, and, as a result, writing the partitions now involve many disk seeks over the partition files.

An interesting point to note from Figures 13 and 15 is that the PBSM and the R-tree-based join algorithm have the same elapsed time for performing the refinement step. For PBSM, the refinement cost constitutes about 45% of the overall join cost, and for the R-tree-based algorithm, the refinement cost is about 23% of the overall join cost. For performing the refinement step, which in this case requires examining two polylines for intersection, a plane-sweeping algorithm was used. Without the plane-sweep algorithm, the cost of the refinement step increases by 62%.

Comparison Using the Sequoia Data

Next, the performance of the algorithms was compared using the Sequoia data set. Figure 16 shows the result of this comparison. For this data set, PBSM is 13–27% faster than the R-tree-based join and 17–114% faster than
the Indexed Nested-Loops join.

![Graph showing execution time in seconds vs buffer pool size](image)

Figure 16: Sequoia Data Set.

Figures 19 through 18 provide detailed cost breakdowns for all the algorithms. In particular, for this data set, the cost of the refinement step is a dominating factor for both the PBSM and the R-tree-based join (contributing about 79% to the overall PBSM join cost, and 68% to the overall R-tree-based join cost). The refinement step for this query involves checking if an island polygon is contained in a landuse polygon. This check for containment is currently implemented in Paradise using a naïve $O(n^2)$ algorithm ($n$ is the number of points in a polygon). There are a number of techniques for reducing the cost of this part of the join [BKSS94] (by an order of magnitude in many cases). These techniques rely on using as a filter in the refinement step, extra information that is precomputed and stored along with each spatial feature. As an example, each polygon could store its minimum bounding rectangle (MBR), and a maximal enclosed rectangle (MER)—which is a rectangle that is fully contained in the polygon. Then, during the refinement step to determine if polygon $p_1$ is contained in polygon $p_2$, the MBR of $p_1$ could be examined for containment in the MER of $p_2$. If this containment holds, $p_1$ is guaranteed to lie within $p_2$, and we can skip further processing. If these techniques were implemented, the relative performance of the PBSM algorithm would improve.

**Summary of the No Pre-existing Index Case**

In summary, overall the PBSM algorithm has better performance than the R-tree-based and the Indexed Nested-Loops based algorithms. When the sizes of the two inputs differ significantly, the Indexed Nested-Loops performs better than the R-tree-based algorithm. Finally, the performance of all the algorithms improve if the join inputs are clustered.

---

3For the Indexed Nested-Loops join the refinement step cost is part of the index probe cost.
2.4.5 Joins in the Presence of Pre-existing Indices

In this section, we investigate the performance of the spatial join algorithms when one or both the inputs to the join already have an index. In this experiment, when one index exists, the Index Nested Loops probes that index, whereas the R-tree-based join algorithm builds an index on the other input and proceeds to “join” the indices. When both indices exist, the Index Nested-Loops probes the smaller index, while the R-tree-based join skips building any indices.

The results of this experiment are shown in Figures 20 and 21. When indices pre-exist on both the inputs, the R-tree-based algorithm has the best performance. Since building an index on the smaller input is not very expensive, the R-tree-based algorithm also has the best performance when an index exists only on the larger input (in the graphs, compare PBSM, Rtree-1-LargeIdx and INL-1-LargeIdx). When an index exists only on the larger input, the Indexed Nested-Loops encounters many buffer misses while probing the index. Even if we chose to build an index on the smaller input and probe it, the index probing cost itself is still greater than the R-tree join cost (compare Rtree-1-LargeIdx and INL-1-SmallIdx).

In the last case, when an index exists only on the smaller input, the PBSM join performs better than the R-tree and the Indexed Nested Loops based joins (in the Figures 20 and 21 compare PBSM with Rtree-1-SmallIdx and INL-1-SmallIdx). For small buffer pool sizes, when joining Hydrography with Roads (Figure 20), the R-tree-based algorithm (labeled as Rtree-1-SmallIdx) performs better than Indexed Nested-Loops (labeled as INL-1-SmallIdx). However, as the buffer pool size increases with respect to the index size, the performance of Indexed Nested-Loops improves rapidly, outperforming the R-tree-based join for large buffer pool sizes in Figure 20, and for all buffer pool sizes in Figure 21.

The performance of the algorithms using the clustered TIGER data qualitatively matched with the results for the non-clustered case, while the performance of the different algorithms using the Sequoia data qualitatively matched the results shown in Figure 20.
In summary, if an index exists on the larger input, or if both inputs have a pre-existing index, then the R-tree-based join algorithm has the best performance, and if an index exists only on the smaller input, then the performance of the PBSM algorithm is better.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Component of the Algorithm</th>
<th>24MB Buffer Pool</th>
<th>8MB Buffer Pool</th>
<th>2MB Buffer Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total Cost</td>
<td>I/O Cost</td>
<td>I/O Contribution</td>
<td>Total Cost</td>
</tr>
<tr>
<td>PBSM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition Road</td>
<td>155.6</td>
<td>35.4</td>
<td>22.7%</td>
<td>153.1</td>
</tr>
<tr>
<td>Partition Hyd</td>
<td>42.5</td>
<td>15.5</td>
<td>36.5%</td>
<td>46.2</td>
</tr>
<tr>
<td>Merge Partitions</td>
<td>114.6</td>
<td>16.3</td>
<td>14.2%</td>
<td>94.9</td>
</tr>
<tr>
<td>Refinement</td>
<td>226.3</td>
<td>62.8</td>
<td>27.8%</td>
<td>297.4</td>
</tr>
<tr>
<td>TOTAL</td>
<td>539.0</td>
<td>130.0</td>
<td>24.1%</td>
<td>591.6</td>
</tr>
<tr>
<td>R-Tree Join</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Build Hyd. Index</td>
<td>125.0</td>
<td>11.4</td>
<td>9.1%</td>
<td>154.0</td>
</tr>
<tr>
<td>Build Road Index</td>
<td>649.5</td>
<td>118.8</td>
<td>18.3%</td>
<td>679.6</td>
</tr>
<tr>
<td>Join Index</td>
<td>74.0</td>
<td>32.3</td>
<td>43.6%</td>
<td>91.7</td>
</tr>
<tr>
<td>Refinement</td>
<td>220.5</td>
<td>64.1</td>
<td>29.1%</td>
<td>296.4</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1069.0</td>
<td>226.6</td>
<td>21.2%</td>
<td>1221.7</td>
</tr>
<tr>
<td>NL-Idx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Build Hyd. Index</td>
<td>119.7</td>
<td>12.4</td>
<td>10.4%</td>
<td>150.9</td>
</tr>
<tr>
<td>Probe Index</td>
<td>925.0</td>
<td>120.7</td>
<td>13.0%</td>
<td>1137.3</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1044.7</td>
<td>133.1</td>
<td>12.7%</td>
<td>1288.2</td>
</tr>
</tbody>
</table>

Table 4: Detailed Cost breakdown, TIGER Data. Join Roads with Hydrography (All times are in seconds)

### 2.4.6 CPU Costs

We now examine the CPU and the I/O costs involved in the spatial join algorithms. Table 4 shows the I/O costs incurred when joining the Road and the Hydrography data from the TIGER data set. Note that, except for
the first component in each join algorithm, every component starts out with some dirty pages left behind in the buffer pool by the previous component. Table 4 shows that, for all the algorithms, the CPU costs dominate the I/O costs (by a large amount in most cases). The reason is two folds. First, performing spatial operations such as probing an R*-tree index, joining partitions using a plane-sweep algorithm, and spatial sorting during bulk loading an index are computationally intensive. Second, the SHORE storage manages works hard at minimizing the I/O costs. Whenever a dirty page has to be flushed to the disk, the storage manager forms a sorted list of all the dirty pages in the buffer pool, and tries to find pages that are consecutive on the disk. These pages are then written to the disk. CPU costs are a dominating factor for the spatial joins on the clustered TIGER and the Sequoia data sets too. These costs are shown in Tables 5 and 6 respectively.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Component of the Algorithm</th>
<th>24MB Buffer Pool</th>
<th>8MB Buffer Pool</th>
<th>2MB Buffer Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>I/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>I/O Contribution</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBM</td>
<td>Partition Road</td>
<td>155.4</td>
<td>38.3</td>
<td>24.6 %</td>
</tr>
<tr>
<td></td>
<td>Partition Hyd</td>
<td>42.0</td>
<td>11.1</td>
<td>26.4 %</td>
</tr>
<tr>
<td></td>
<td>Merge Partitions</td>
<td>101.6</td>
<td>16.7</td>
<td>16.4 %</td>
</tr>
<tr>
<td></td>
<td>Refinement</td>
<td>229.4</td>
<td>62.6</td>
<td>27.3 %</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>528.4</td>
<td>128.7</td>
<td>24.4 %</td>
</tr>
<tr>
<td>R-Tree Join</td>
<td>Build Hyd. Index</td>
<td>84.3</td>
<td>10.7</td>
<td>12.7 %</td>
</tr>
<tr>
<td></td>
<td>Build Road Index</td>
<td>366.4</td>
<td>56.0</td>
<td>15.3 %</td>
</tr>
<tr>
<td></td>
<td>Join Indices</td>
<td>72.4</td>
<td>31.2</td>
<td>43.1 %</td>
</tr>
<tr>
<td></td>
<td>Refinement</td>
<td>213.8</td>
<td>122.0</td>
<td>57.1 %</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>736.9</td>
<td>219.9</td>
<td>29.8 %</td>
</tr>
<tr>
<td>NL-Idx</td>
<td>Build Hyd. Index</td>
<td>82.0</td>
<td>11.1</td>
<td>13.5 %</td>
</tr>
<tr>
<td></td>
<td>Probe Index</td>
<td>860.3</td>
<td>122.0</td>
<td>14.2 %</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>942.3</td>
<td>133.1</td>
<td>14.1 %</td>
</tr>
</tbody>
</table>

Table 5: Detailed Cost breakdown, Clustered Tiger Data. Join Roads with Hydrograph (All times are in seconds)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Component of the Algorithm</th>
<th>24MB Buffer Pool</th>
<th>8MB Buffer Pool</th>
<th>2MB Buffer Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>I/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>I/O Contribution</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBM</td>
<td>Partition Islands</td>
<td>7.9</td>
<td>2.1</td>
<td>26.6 %</td>
</tr>
<tr>
<td></td>
<td>Partition Polygons</td>
<td>27.1</td>
<td>10.0</td>
<td>36.9 %</td>
</tr>
<tr>
<td></td>
<td>Merge Partitions</td>
<td>12.8</td>
<td>1.0</td>
<td>7.8 %</td>
</tr>
<tr>
<td></td>
<td>Refinement</td>
<td>193.3</td>
<td>34.5</td>
<td>17.8 %</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>241.1</td>
<td>47.6</td>
<td>19.7 %</td>
</tr>
<tr>
<td>R-Tree Join</td>
<td>Build Poly. Index</td>
<td>54.2</td>
<td>9.7</td>
<td>17.9 %</td>
</tr>
<tr>
<td></td>
<td>Build Isl. Index</td>
<td>22.7</td>
<td>5.8</td>
<td>25.6 %</td>
</tr>
<tr>
<td></td>
<td>Join Indices</td>
<td>28.5</td>
<td>0.1</td>
<td>0.4 %</td>
</tr>
<tr>
<td></td>
<td>Refinement</td>
<td>200.5</td>
<td>34.0</td>
<td>17.0 %</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>305.9</td>
<td>49.6</td>
<td>16.2 %</td>
</tr>
<tr>
<td>NL-Idx</td>
<td>Build Isl. Index</td>
<td>19.2</td>
<td>3.5</td>
<td>18.2 %</td>
</tr>
<tr>
<td></td>
<td>Probe Index</td>
<td>263.8</td>
<td>40.2</td>
<td>15.2 %</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>283.0</td>
<td>43.7</td>
<td>15.4 %</td>
</tr>
</tbody>
</table>

Table 6: Detailed Cost breakdown, Sequoia Data. Join Polygons with Islands (All times are in seconds)
2.4.7 Effect of Spatial Partitioning Function Parameters on PBSM

In this section, we investigate the effect of two spatial partitioning function parameters on the performance of PBSM. The first parameter is the number of tiles used in the spatial partitioning function, and the second parameter is the number of partitions that an input is broken into. As shown in Section 2.3.4, increasing the number of tiles and decreasing the number of partitions produces more uniform partitions. However, with both these techniques the replication overhead, which is the increase in the number of tuples due to replication, increases. We note that “over-partitioning” (dividing the input into many more partitions than required) has been used in the Adaptive hash join algorithm for dealing with partition skew [ZG90].

For the numbers presented in this section, the buffer pool size is fixed at 8MB.

<table>
<thead>
<tr>
<th># Partitions</th>
<th>256 tiles</th>
<th>1024 tiles</th>
<th>4196 tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P-Road</td>
<td>P-Hyd</td>
<td>Merge P-Road</td>
</tr>
<tr>
<td>4</td>
<td>154.0</td>
<td>45.6</td>
<td>98.1 297.7</td>
</tr>
<tr>
<td>8</td>
<td>162.6</td>
<td>49.9</td>
<td>86.3 298.8</td>
</tr>
<tr>
<td>16</td>
<td>182.7</td>
<td>54.2</td>
<td>85.2 322.1</td>
</tr>
<tr>
<td>32</td>
<td>197.9</td>
<td>61.9</td>
<td>84.5 344.3</td>
</tr>
</tbody>
</table>

Table 7: TIGER Data, Join Road with Hydrography (All times are in seconds)

Table 7 shows the effect of the two parameters when joining the Road data with the Hydrography data from the TIGER data set (refer to Section 2.4.3 for the detailed description of this data set). This table only shows the cost of the filter step of PBSM. The cost of the refinement step for this query is around 300 seconds for all the cases. In the table, P-Road is the cost of partitioning the Road relation, P-Hyd is the cost of partitioning the Hydrography relation, and Merge is the cost of merging the partitions.

First, we observe the effect of changing the number of tiles. Reading across the rows in Table 7, we observe that as the number of tiles increases, the partitioning costs increase. This behavior is seen because increasing the number of tiles increases the replication overhead, which requires that more tuples be written to the disk. For the merging costs, increasing the number of tiles has two conflicting effects. On one hand, increasing the number of tiles increases the replication overhead, which leads to a higher merge cost. At the same time, increasing the number of tiles balances the partitions better, and this tends to decrease the merge cost. The merge step involves sorting the elements (refer to Section 2.3.1) that are being merged, and this cost being non-linear \( O(n \log n) \), decreases if the partitions are better balanced. Overall, we observe that increasing the number of tiles increases the cost of filter step by a small amount (at most 3% in Table 7). If we include the cost of the refinement step, then this effect becomes even smaller.

Second, we observe that as we increase the number of partitions, the partition costs increase. There are two reasons for this. The first is the higher replication overhead, and the second is the larger number of partition files that are created on the disk. The PBSM algorithm does not manage any of the partition buffers; it simply writes tuples to appropriate partition files, and relies on the SHORE storage manager to flush pages of the partition files to disk. When writing a dirty page to the disk, the storage manager groups together pages that are consecutive on the disk, and writes them out together. However, writing out a group of pages might require one disk seek. With a larger number of partition files, the number of these disk seeks increases. Increasing the number of partitions
decreases the merge cost because each partition is now much smaller, and the non-linearity in the merge cost reduces this cost.

We also examined the behavior of PBSM with two other data sets: Clustered TIGER data, and the Sequoia data (once again, refer to Section 2.4.3 for the detailed description of these data sets). The results for this are similar and are shown in Tables 8 and 9.

<table>
<thead>
<tr>
<th># Partitions</th>
<th>256 tiles</th>
<th>1024 tiles</th>
<th>4196 tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P-Road</td>
<td>P-Hyd</td>
<td>Total</td>
</tr>
<tr>
<td>4</td>
<td>156.3</td>
<td>45.4</td>
<td>99.4</td>
</tr>
<tr>
<td>8</td>
<td>157.0</td>
<td>46.3</td>
<td>89.0</td>
</tr>
<tr>
<td>16</td>
<td>161.4</td>
<td>50.3</td>
<td>84.6</td>
</tr>
<tr>
<td>32</td>
<td>166.7</td>
<td>53.2</td>
<td>83.6</td>
</tr>
</tbody>
</table>

Table 8: Clustered TIGER Data, Join Road with Hydrography (All times are in seconds)

<table>
<thead>
<tr>
<th># Partitions</th>
<th>256 tiles</th>
<th>1024 tiles</th>
<th>4196 tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P-Island</td>
<td>P-Poly</td>
<td>Total</td>
</tr>
<tr>
<td>1</td>
<td>8.3</td>
<td>25.2</td>
<td>16.3</td>
</tr>
<tr>
<td>4</td>
<td>8.5</td>
<td>26.2</td>
<td>16.9</td>
</tr>
<tr>
<td>16</td>
<td>14.3</td>
<td>30.2</td>
<td>15.3</td>
</tr>
</tbody>
</table>

Table 9: Sequoia Data, Join Polygons with Islands (All times are in seconds)

### 2.5 Conclusions

In this chapter the Partition-Based Spatial-Merge (PBSM) Join, a new algorithm for performing spatial join, has been described. This algorithm does not require any indices on the joining attribute of the two inputs. Such a situation could arise if both the inputs to the join are intermediate results in a complex query, or in a parallel environment where the inputs have been dynamically redistributed. The algorithm uses an efficient computational geometry based plane-sweeping technique for performing the join. If the inputs to the algorithm are too large to fit in memory, then a spatial partitioning function is used to partition the inputs into chunks that can fit in memory.

This chapter also describes the results of a comprehensive performance study that is based on actual implementation of three spatial join algorithms in Paradise, a database system for handling GIS applications. The three algorithms are: the traditional indexed nested loops algorithm, a previously proposed algorithm that uses spatial indices on both the inputs to evaluate the join, and the PBSM algorithm. The performance comparison, using real data sets, show that the PBSM algorithm is more efficient when neither of the inputs to the join have a spatial index (ad-hoc spatial joins). When an index exists only on the smaller input, the PBSM algorithm still performs better than the other algorithms. As a rule of thumb, if the index building time is more than than the join time, then PBSM outperforms the other algorithms. In the case where the index building time is small relative to the join time, then the R-tree-based algorithm has better performance. A database system that needs to efficiently support ad-hoc spatial join operations would benefit from using the PBSM algorithm.
Chapter 3

Spatial Aggregates

This chapter deals with the spatial aggregate operation. The bulk of this chapter focuses on the most commonly used spatial aggregate operation, namely, the nearest-neighbor search. First, the previous work in this area is presented. This is then followed by the description of a simple algorithm for evaluating the nearest-neighbor search. Finally, the results from implementing this algorithm are presented.

3.1 What is a Spatial Aggregate?

Frequently, a spatial database system is used to answer queries like “find the river that is closest to a given city”. Conceptually, evaluating such a query involves looking at each river feature, computing the distance of the river to the chosen city, and producing as its result the river is closest to the city. This operation of selecting a spatial feature from a set of spatial features based on some spatial relationship between the features in the set to some reference feature is called a spatial aggregation. Other examples of spatial aggregation include queries like “find the river that is farthest from the city and within the state boundary of the city”, or “find the closest river to the north of the city”.

3.2 Previous Work

The first spatial aggregate query described in the previous section is a variant of the n nearest-neighbor search problem that involves finding the n nearest-neighbors to a given point. Most of the work on spatial aggregation has focused on finding the n nearest-neighbors. Most of these algorithms make use of a spatial index to speed up evaluation of this operation. Samet [Sam90] proposed an algorithm to find the n nearest-neighbors using a PM Quad Tree index [Tam81]. This algorithm first traverses the quadtree from the root to find the leaf that contains the query point. Then, it backtracks the path that it just traversed, and uses a heuristic to search the subtrees that potentially contain the n nearest-neighbors. An algorithm for solving the nearest-neighbor problem with k-d trees [Rob81] was proposed by Friedman, Bentley and Finkel [FBF77]. This algorithm was later refined by Sproull [Spr91]. Roussopoulos, Kelley and Vincent [RKV95] have proposed a branch-and-bound search algorithm for finding the nearest neighbors using R-trees [Gut84]. This algorithm traverses down the R-tree following all the branches that might contain one of the n-neighbors. A heuristic is used to reduce the number of branches that are traversed.

1If multiple rivers have the same distance to the city, one of them is randomly selected by this operation.
3.3 Nearest Neighbor Search Using an Index Independent Algorithm

All the previous algorithms for solving the nearest neighbor problem are tied to an indexing structure. We now propose a very simple algorithm that can be applied with any spatial index. We assume that a spatial index has a primitive that allows the retrieval of all spatial objects that are contained within a box. Using only this primitive the search for the nearest neighbor can be conducted as follows.

1. Start with a circle using some initial guess for the radius. Take the bounding box of the circle and probe the index with the box.

2. Examine the results of the index probe and check if any of the result tuples overlaps with the circle. If there are no result tuples within the radius of the circle, then expand the radius of the circle using some heuristic. For example, increase the radius to get a circle that has twice the area.

3. Repeat steps 1 and 2 till we have a circle that returns at least one tuple.

4. Fetch all the tuples that are returned by the index search, and compute the actual distance of the spatial feature in the tuple to the reference point. Keep the tuple with the closest spatial feature.

This algorithm can be refined in a number of ways.

- If the “probing” circle expands beyond the boundary of the universe, then the index scan can be changed to a file scan (which is faster for scanning the whole relation).

- If an index probe returns a large number of tuples, then in step 2, instead of expanding the circle, we can reduce the radius of the circle and probe the index. Of course, if the probe of the index with this smaller circle does not produce any tuples, then we can go back to the previous circle. Another way of fine tuning this algorithm is to change the rate at which the radius of the circle is increased or decreased.

- This algorithm can also be modified to find the n nearest-neighbors. To find the n nearest-neighbors we change the stopping criteria (Step 3) to stop when the index probe returns at least n tuples. Then, in Step 4, we retrieve the actual tuples and order them by their distance to the reference point, keeping only the first n tuples.

- The above algorithm can be modified if the reference feature is something other than a point. For example, consider Figure 22 where the reference feature is a polygon. To find the closest feature to the polygon, we can construct a buffer region around the polygon and probe the index with this buffer polygon. Similar to expanding the circle, we can expand the buffer region around the polygon till we find the answer.

- The buffer polygon approach can also be used in other situations, such as answering aggregate queries like “find me the closest feature to the north of this feature”. In this case the buffer polygon would be constructed so that it does not include any region to the south of the spatial feature.
3.4 Index Independent Algorithm For General Spatial Aggregation

The algorithm outlined above can be generalized for other types of spatial aggregates. Take for example the following query: “find the farthest river to a city that is within the state boundary”. Similar to the strategy employed above, this query can be evaluated by using an index on the city relation. Consider Figure 23 that shows a city and the state polygon around it. To evaluate this query, a small buffer region can be constructed that lies inside the state. The index can then be probed to find rivers that overlap with this buffer region. Note that the buffer region that must be probed is a swiss-cheese-polygon. One way to probe the index is to use the bounding box corresponding to the buffer region. However, such a probe would include all the rivers in the state. If there are many such rivers, this may not be the most efficient way of evaluating the query. Another possible alternative for retrieving all the rivers in the buffer region is to approximate the buffer regions by a series of “probe” boxes. For example, in Figure 24 the buffer region is approximated by four rectangular probe boxes. The spatial index can then be examined to find river tuples that overlap with each of these probe boxes. Matches to these probes can then be examined to determine if the river actually lies in the buffer region. If the first buffer region does not produce any result, then the buffer region can be iteratively made smaller, stopping when the buffer region degenerates into a point.

3.5 Spatial Aggregation with Joins

We now consider a slight variant of the spatial aggregation problem. Consider a database with the following two relations:

Relation ToxicDump (name: String, location: Point);
Relation City (name: String, location: Point, population: Integer);

Now consider the query “find the closest toxic waste dump to every large city with a population greater than one million”. Conceptually, this query requires evaluating a cross product of two relations (City and ToxicDump),
followed by a spatial aggregate with a group by on the city. A system that does not have any special algorithms for handling spatial aggregates would probably evaluate the query using the operator tree shown in Figure 25. In this case, a select operator first selects all large cites that get joined with the tuples from the ToxicDump relation. This join is a cartesian product and is evaluated using nested loops. The result of the join is run through an aggregate operator that groups by the city and computes the closest ToxicDump in each group.

A more efficient way of evaluating this query is to generalize the spatial aggregate algorithm described in Section 3.3 to a “join with aggregate” operator. For each City tuple, this algorithm uses the index on the ToxicDump relation and runs the aggregate algorithm described in Section 3.3. The operator tree for this is shown in Figure 26.

3.6 Performance Evaluation

This section contains the results of implementing the spatial aggregate algorithm described in the previous section in the Paradise [DKL+94, PYK+97] geo-spatial database system. The hardware platform for this performance evaluation was a Solaris 2.5 machine with dual 133 MHz Pentium processors, and 128 MBytes of memory. One Seagate Barracuda disk drive (ST32500WC) with a storage capacity of 2.1 GBytes was used for storing the data. The logs for the database was kept on a separate Seagate disk. For this experiment, Paradise was configured to use a 32 MByte buffer pool. Although this is a small buffer pool relative to the 128 MByte of physical memory available, Paradise does much of its query processing outside the buffer pool in dynamically allocated memory. The maximum process size we observed during benchmark execution was about 95 MBytes. Thus, no swapping occurred.

The data set used in this study is the drainage and the cities data set from the Digital Chart of the World [DCW92]. The characteristics of the data is shown in Table 10. Each Drainage tuple has a polyline attribute that describes the drainage feature, and each Cities tuple has a point attribute that describes the geo-location of the city. An
R*-tree index was built on the drainage relation, and the build time for this index was 1080 seconds.

<table>
<thead>
<tr>
<th>Relation</th>
<th># of Objects</th>
<th>Total Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drainage</td>
<td>1.73 M</td>
<td>300 MB</td>
</tr>
<tr>
<td>Cities</td>
<td>0.25 M</td>
<td>20 MB</td>
</tr>
</tbody>
</table>

Table 10: DCW Cities and Drainage Data Set

On this data set, the system was asked to find closest (using euclidean geometry) drainage features to a bunch of cities that were selected randomly from the entire set of cities. The query was evaluated in two different ways, namely, using the nested-loops and the “join with aggregate” operator. In the experiment, the number of cities that were selected was varied from 1 to 100.

The results of this experiment are shown in Figure 27. Note that the y axis in this Figure is logarithmic. As shown in the figure, use of the “join with aggregate” operator produces a major improvement in performance. Using the index to restrict looking at only a small portion of the drainage relation makes the “join with aggregate” operator much more efficient.

The figure also shows that the performance of the nested-loops join algorithm degrades rapidly as the number of selected cities increases. This behavior is because the size of the intermediate result produced by the nested loops join operator in Figure 25 increases linearly with the number of selected cites. Consequently, the cost of producing the intermediate result, and the cost of the final aggregate operator increases proportionately. In contrast, the algorithm that uses the “join with aggregate” operator (see Figure 26) only requires additional index probes as the number of cities is increased.
3.7 Conclusions

In this chapter, we have described a simple algorithm for evaluating the spatial aggregate operation. This algorithm uses a spatial index to speed up the processing, but unlike previous algorithms, this algorithm is not tied to a specific indexing structure. This algorithm can be generalized to a “join with aggregate” operator to efficiently evaluate spatial aggregates on groups of tuples. Use of the “join with aggregate” operator can dramatically reduce the time for evaluating a spatial aggregate operation.
PART II: Parallel Spatial Database Techniques
Chapter 4

Declustering Spatial Data

In the last few decades, there has been a dramatic increase in the volume of the spatial data that has become available to applications. USGS, for example, distributes the TIGER data set that contains cartographic data for the United States. The size of the entire data set at a resolution of 1:100,000 is 19 GB. In the near future, we will probably see even larger data sets. High-resolution satellite images are now becoming available for commercial purposes. A variety of applications are developing around these high resolution satellite images. Using image processing techniques, a satellite image can be decomposed into many polygons, points and polylines representing features in the image. These extracted features can then be put into a database system and queried. This process can easily generate very large spatial data sets, and spatial database systems must be prepared to efficiently handle very large volumes of data. In the relational world, parallelism has successfully been employed to store and query large volumes of relational data. It is natural then, to try using parallelism to store and query large spatial data sets.

Beginning with this chapter, the second part of this thesis explores the use of parallelism to store and query spatial data, with an emphasis on the use of a shared-nothing database system. In a parallel shared-nothing system, the main source of parallelism is partitioned parallelism [DG92]. Partitioned parallelism is achieved by declustering the data across multiple nodes in the system, and then running operators at each of these nodes. There are two main requirements for achieving effective parallelism. First, good data declustering techniques are required to evenly distribute the data across the nodes in the parallel system. Second, the operators must be designed such that an operator running at a particular node accesses only the data stored locally. Naturally, in an attempt to build a parallel spatial database engine, one must first explore various declustering techniques that can be used to distribute spatial data across nodes in a parallel database system, and then explore various algorithms for evaluating spatial operations.

This chapter examines declustering strategies for distributing spatial data across various nodes in a parallel shared-nothing database system. The declustering strategies that are considered in this chapter are based on statically partitioning the underlying spatial domain and then mapping these partitions to nodes in the parallel system.

4.1 Spatial Data and Relational Parallelism

At first, it might seem that the techniques developed for parallel relational databases could be applied to spatial databases too. However, spatial data is fundamentally different from relational data; spatial data, such as polygons and polylines, actually spans space. A tuple in the relational world can be viewed as a point in a
multiple dimension space. If we draw a hyper-plane through this multidimensional space, the point lies on the plane or on one of the sides of the plane. On the other hand, a polygon in a multidimensional space could have portions of it on both sides of the hyper-plane (and of course, intersect the hyper-plane too). This fundamental difference gives rise to many interesting differences that necessitate novel techniques for handling parallelism for spatial query processing.

4.2 Related Work

Parallel spatial databases are a relatively new area, and, in the last few years some attention has been directed towards various aspects of a parallel spatial database system. This section reviews the related work in declustering spatial data.

Tan and Yu [TY95] have proposed a few techniques for declustering spatial data, and have evaluated the effect of these declustering techniques on spatial selections. The data used in the study is synthetically generated, and is uniformly distributed in the space. Consequently, data distribution skew is not considered at all. In [AOT+95] the authors propose a spatial semi-join operator for joining spatial data from two distributed sites in a distributed spatial database system. No parallel evaluation technique is explored in the paper. Kamel and Faloutsos [KF92] have examined the use of parallelism to accelerate the performance of spatial selections. They examined techniques to distribute the leaves of an R-tree [Gut84] across multiple disks in a system, and evaluated various placement policies for the leaf pages. The parallel architecture that was targeted has one processor with multiple disks attached to it, and the effectiveness of the declustering strategy is examined only for a simple environment where the queries consist only of simple spatial selections. This idea was later extended by Koudas, Faloutsos and Kamel [FKK96] to decluster spatial data on a shared-nothing architecture. One node in the system was dedicated for performing the bookkeeping associated with mapping the leaves of the R-tree to the other nodes in the system. Again, the focus was on an environment with spatial selection queries.

Recently, Zhou, Abel and Truffet [ZAT97], have examined data partitioning mechanism for parallel spatial join processing. First, the authors show that for parallel spatial joins, a static partitioning function is superior to a dynamic partitioning function. Then, the authors propose a parallel spatial join algorithm that uses a static partitioning function. Similar to the spatial partitioning function used in [PD96] (see Section 2.3.4), the data is partitioned by first dividing the space into cells, and then the cells are mapped to partitions. In As in [PD96] if a spatial object overlaps cells that are mapped to different partitions, the object is inserted into multiple partitions. A spatial join algorithm is developed based on this partitioning strategy. A major focus of the paper is on tuning the spatial join algorithm to handle data skew [KO90, WDJ91], and balancing the workload across all the nodes in the system. The performance of the algorithm is evaluated using a 30MB data set by simulating a parallel environment on a SunSPARC 10 workstation. All data is always assumed to fit in memory, and a single query is used for the performance evaluation. One simulation model is used to calculate the CPU cost of executing the query, and another simulation model is used to calculate the network message cost. The total cost of the query is calculated by adding these costs. The model does not account for overlap of communication and CPU processing, or contention for network resources.
4.3 Spatial Declustering Techniques

4.3.1 Declustering Spatial Data Using Replication

One way of declustering a relation on one of its spatial attribute is as follows. First, take the universe of the spatial attribute on which the relation is being declustered. The universe of a particular spatial attribute is defined as the minimum rectangle that covers (encloses) that spatial attribute for all the tuples in the relation. After determining the universe (this information could be part of the statistics stored in the database catalogs), divide the universe into as many regions as there are nodes (i.e., processors with disks) for declustering the spatial relation. Now, map the regions to the nodes using some assignment function. Tuples whose declustering spatial attribute is completely enclosed within the boundary of a region are placed at the node corresponding to that region. Tuples whose declustering spatial attribute overlaps multiple regions need special handling. As an example, consider Figure 28 where there are 4 nodes in the system. The universe of the polygon attribute on which the relation is being declustered is broken up into regions 0 to 3. These regions are mapped to nodes using an identity assignment function (i.e. region x is mapped to node x). Two polygon attributes, labeled A and B, are shown in the figure. Since polygon B is completely enclosed within region 1, the tuple corresponding to polygon B is placed at node 1. Polygon A, however overlaps regions 0 and 2, and one way of dealing with this tuple is to replicate it at both nodes 0 and 2. To see the need for replication, consider another relation that has tuples with an attribute containing a polyline corresponding to a highway (see Figure 29).

Assume that this relation is being declustered using the same region boundary as the polygon relation above. Highway I-90 in Figure 29 is mapped to node 0. Now consider the problem of finding all polygons that overlap with highways. This query would make sense if for example the polygons were corn fields and we wanted to find all highways that pass through corn fields. Consider the processing of the highway I-90 at node 0 (where it resides), and the processing of highway I-80 at node 2. To detect that polygon A overlaps both highway I-90 and highway I-80, both nodes 0 and 2 must somehow know that polygon A spans the space covered by these two highways. One way of ensuring this is to replicate the polygon A at both the nodes (as proposed by Zhou, Abel and Truffet [ZAT97]). The problem with this declustering strategy is that it might produce a skewed distribution of data. As an example, consider declustering a relation in a four-node system where most of the declustering spatial attributes are located in the upper left corner (see Figure 30). If the universe is simply divided into 4
uniform regions, then most of the tuples will get mapped to node 0. Since very few tuples get mapped to the other nodes, the resulting declustering of tuples is highly skewed. In a parallel system, such skew can cause severe performance problems [WDJ91, KO90]. To mitigate this problem, we divide the universe into many small regions called *tiles*. Then, the tiles are assigned unique tile numbers by walking through the tiles in a row major or column major order. The tiles are then mapped to nodes by either hashing on the tile number to get a node number, or by assigning tiles to nodes using a round robin scheme. As an example, Figure 31 shows the same relation as in Figure 30 being declustered across 4 nodes using 16 tiles, with the tiles being mapped to nodes using some hash function. This declustering strategy is similar to the spatial partitioning function that is used internally in the PBSM spatial join algorithm (see Section 2.3.4). There it was shown that hashing is superior to round robin. Consequently, in this section we only consider hashing.

![Figure 30: Data Distribution Skew](image)

![Figure 31: Tiling to Reduce Data Distribution Skew](image)

Let us now examine the number of tiles that are needed to get a good data distribution. One expects the data distribution to improve as the number of tiles is increased. With more tiles, dense regions, such as the upper left region in Figure 30, get mapped to multiple tiles, which, in turn, get mapped to different nodes. The cost that we pay for obtaining this better distribution is an increase in the replication cost. As the number of tiles increases, more and more tuples will have spatial attributes that overlap tiles that are mapped to different nodes, and this increases the number of tuples that get replicated. To quantify these tradeoffs, we ran some experiments using the drainage data set from the data provided by DCW [DCW92]. This data set describes drainage features, like rivers, streams, canals, etc., for the entire world. This relation has 1.7 million tuples, and the size of the relation is about 300 MB. To measure the effectiveness of the declustering strategy, this relation was spread across an 8-node and an 128-node configuration. In each of these configurations, the number of tiles used in the declustering strategy was varied. To measure the effectiveness of the spatial declustering mechanism, we viewed the number of tuples across the nodes as a distribution and measured the coefficient of variation of this distribution. A good distribution would have a small value for the coefficient of variation (as a reference point, the exponential distribution has a coefficient of variation of 1). The result of declustering this data set is shown in Figure 32. The figure shows that increasing the number of tiles produces a better distribution. As expected, a system with fewer nodes has a better data distribution. Compare the 8-node and 128-node configurations in Figure 32).

Figure 33 shows the replication overhead for the same data set. The replication overhead is measured in terms of the additional disk space that is required to store the declustered relation. As shown in the figure, the replication overhead is quite small for this data set. However, the replication overhead could change dramatically if
the data set has different characteristics. For example, consider the landuse data set from the Sequoia benchmark [SFGM93]. This data set is 32 MB in size and contains 60K landuse polygons for the state of California and Nevada. First, examine Figure 34 which shows the data distribution skew for this data set. Similarly to the DCW landuse data set (see Figure 32), increasing the number of tiles produces a better data distribution. However, the space overhead due to replication is very different for the two data sets. Figure 35 shows the extra disk space that is required to store the declustered Sequoia landuse relation. As shown in the figure, the space overhead due to replication can be very high. For example, in the 5000 tiles 128 node configuration, the size of the declustered relation on disk increases by 300%. Figure 36 shows the number of tuples that are replicated for the Sequoia landuse data set. Examining the 128 node configuration in the figure shows that with 5000 tiles about 30% of the tuples get replicated. It is these 30% of the tuples that cause the space overhead to increase by 300%. This is because the polygons that tend to get replicated are those that cover larger areas of space. In turn, these polygons require a large number of points to represent their geometry, and, as a result, require a proportionately large amount of disk space. Consequently, replicating them causes a dramatic increase in the disk space overhead.

4.3.2 Partial Spatial Surrogates

To solve the problem of high replication overhead, we can employ the following strategy. When the declustering spatial attribute of a tuple overlaps tiles that are mapped to multiple nodes, we pick one of the nodes as the home node. The entire tuple is stored only at the home node, while all nodes (including the home node) store the global
OID of the tuple and the minimum bounding rectangle (MBR) of that part of the spatial attribute that overlaps the area covered by the node. This replicated MBR is called the fragment box. The fragment box and the OID of the tuple are collectively called the partial spatial surrogate of the declustered spatial attribute, and are stored in a separate relation. Since a partial spatial surrogate requires very little space (about 16 bytes for the MBR, and 16 bytes for the OID), the increase in size of the declustered relation due to replication is quite small. As an example of this declustering strategy, consider Figure 37 which shows a tuple being declustered on a polygon attribute. The universe has been divided into 9 tiles and the tiles have been mapped to 3 nodes. Let node 1 be the home node for the tuple, and let OID-H represent the global OID of the tuple. Then, node 0 will store the fragment box FB0 and OID-H, node 2 will store the fragment box FB2 and OID-H, and node 1, in addition to storing the tuple, will also store the fragment box FB1 and OID-H. A partial spatial surrogate at a particular node represents a conservative approximation of the portion of the spatial attribute that is within the area of the universe assigned to the node. It serves as a signal that if someone is interested in tuples in this portion of space, then they should follow the OID to the node where the tuple actually resides.
To quantify the effectiveness of using partial spatial surrogates, let us revisit the Sequoia polygon data set that was used in Figures 34, 35, and 36. Figure 38 shows the space overhead for this data set using partial spatial surrogates. Comparing Figure 38 with Figure 35 shows that the use of partial spatial surrogates dramatically reduces the space overhead associated with spatial declustering.

4.4 Conclusions

To summarize, in this chapter, two spatial declustering strategies have been proposed. In the first strategy, entire tuple is replicated when the declustering spatial attribute overlaps tiles that are mapped to multiple nodes. For the remainder of this thesis, this declustering strategy is referred to as "decluster using whole tuple replication" (D-W). The other strategy creates partial spatial surrogates when the spatial attribute overlaps tiles that are mapped to multiple nodes. This strategy is referred to as "decluster creating spatial surrogates" (D-PSS). Both D-W and D-PSS require a large number of tiles (few thousands for a few tens of nodes) to get a good data distribution. Spatial sampling techniques [OR93] can also be used to estimate the appropriate number of tiles.

Note that both replicating whole tuples and creating partial spatial surrogates has an associated update overhead. Fortunately, spatial database applications tend to update existing tuples rarely, and hence this form of replication is practical.
Chapter 5

Parallel Spatial Join Algorithms

This chapter examines various algorithms for evaluating the spatial join operation in a parallel shared-nothing spatial database system. First, based on the declustering strategies discussed in the previous chapter, the design space of the parallel spatial join algorithms is laid out. This is followed by the development of an analytical cost model for these algorithms. Finally, the performance of these algorithms is compared based on an actual implementation.

5.1 Execution Phases of Parallel Spatial Joins

A parallel spatial join algorithm executes in three phases.

1. Partitioning Phase
2. Join Phase
3. Refinement Phase

In the partitioning phase, the two relations being joined are reclustered on their (spatial) join attributes. If, before the join, one or both relations happen to be declustered on the joining attribute, then this phase is not required. In the context of spatial declustering, the “same” declustering means that the two relations are declustered using the same universe, the same tile boundaries, and the same tile-to-node mapping. Note that when this condition holds, one of the relations could be declustered using the strategy where whole tuples are replicated (D-W) and the other relation could be declustered by creating partial spatial surrogates (D-PSS).

In the join phase, each operator looks at the fragment of the declustered relation residing on its local disks and joins them using any of the centralized join algorithms [Ore86, BKS93, Gün93, HJR97, GS87, LR94, HS95, LR96, PD96]. In this chapter, the local processing is done using the PBSM join algorithm [PD96] (described in Section 2.3).

The final refinement phase is required for two reasons. First, if one of the inputs in the partitioning is reclustered using D-PSS (see Section 4.3.2), then the join phase will use the fragment box for joining the two relations. The fragment box is just an approximation of the spatial object, and, in the final refinement phase, the tuple corresponding to the fragment box is examined to determine if the spatial attribute actually satisfies the join predicate. The second use of the refinement step is to handle the duplicates that may be produced when a relation in the partitioning phase is reclustered using D-W (see Section 4.3.1). For example consider Figure 39, where
the two polygons A and B have overlapping portions at both nodes 0 and 2. With D-W as the redeclustering strategy, both nodes 0 and 2 would have copies of both the polygons, and, consequently, the local join at both these nodes will produce an identical result pair. In the last refinement phase, a distinct operation is used to eliminate one of these result pairs.

Figure 39: Need for Duplicate Elimination

Figure 40: Design Space of Spatial Join Algorithms

5.2 Design Space

Let $R$ and $S$ denote the two relations that are being joined. As mentioned above, if required, in the partitioning phase these relations are redeclustered using either D-PSS or D-W. Based on these alternatives, the design space of the parallel spatial join algorithms is as shown in Figure 40. Algorithm “A” corresponds to the case when both relations are redeclustered using D-PSS. The operator tree for this algorithm is shown in Figure 41. The first two operators (labeled as operators 1 and 2 in Figure 41) redecluster the relations producing partial spatial surrogates. The next operator, Operator 3, joins the partial spatial surrogates producing a candidate set. The candidate set contains a pair of OIDs; one of the OIDs in this pair points to a tuple in the relation $R$, and the other points to a tuple in the relation $S$. This candidate set is redeclustered (by Operator 3) on the node information in OID-$R$. Effectively, this redeclustering sends each candidate to the home node of the $R$ tuple. Operator 4 then “joins” the candidate set with the $R$ tuples. To ensure that the $R$ tuples are read sequentially, this operator sorts the incoming candidates before fetching the tuples from the relation $R$. After this step, the intermediate result is declustered on OID-$S$. This redeclustering is followed by the last operator which “joins” with the relation $S$.

Now consider Algorithm “B” in the design space (see Figure 40). The operator tree for this algorithm is shown in Figure 42. The first two operators (labeled as operator 1 and 2 in Figure 42) redecluster the relations using the D-W declustering policy. Next, a local spatial join operator joins the redeclustered relations. Finally, as mentioned earlier (refer to Figure 39), a distinct operator ensures that the replication does not produce any additional duplicates.

Referring back to Figure 40, now consider Algorithm “C”. This algorithm first redeclusters one of the relation using D-W and the other using D-PSS, and then joins these redeclustered relations. This algorithm is a special case of Algorithm “A” with one less “join on OID” operator.
Figure 41: Join Using Partial Spatial Surrogates For Redeclustering (Algorithm A—Shadow Join)
Figure 42: Join Using Whole Tuple Redeclustering (Algorithm B—Clone Join)
The algorithms in Figure 40 can be adapted if one or both the relations are already declustered on the join attribute. For example, if the relations happen to be declustered using D-W prior to the join, then Algorithm “A” can be run without the redeclustor operators. Along similar lines, if only one of the relation is declustered on D-W prior to the join, then we can either redecluster the other relation using D-W and run Algorithm “A” (without the redeclustor operators), or redecluster the other relation using D-PSS and run Algorithm “C” (without the redeclustor operator).

For the rest of this chapter, we shall refer to Algorithm “A” as the Shadow Join, since this algorithm uses partial spatial surrogates that are like shadows of the actual spatial attribute. Algorithm “B” replicates entire tuples during the redeclustering process, essentially creating clones. Subsequently, we refer to this algorithm as the Clone Join.

The Shadow join algorithm just described is similar to the parallel spatial join proposed by Zhou, Abel and Truffet [ZAT97]. The declustering strategy employed that algorithm is a form of D-PSS. In D-PSS, when a spatial attribute overlaps tiles that are mapped to multiple nodes, the MBR of the spatial attribute is broken up into fragment boxes (refer to Section 4.3.2). The fragment boxes are then sent to the appropriate node. This step ensures that a node only sees the portion of the spatial attribute that is relevant to the space covered at that node. In [ZAT97], when a spatial attribute overlaps tiles that are mapped to multiple nodes, the entire MBR is replicated. This strategy might lead to some wasted processing in the plane-sweep algorithm that is used in the local spatial join (see Section 2.3 for the plane-sweep algorithm). Another difference between Shadow join and the algorithm in [ZAT97] is that Shadow join uses a plane-sweep algorithm for performing the local spatial join. The algorithm in [ZAT97] uses a nested-loops join, and always assumes that all the data fits in main memory.

5.3 Analytical Model

Let us now compare the performance of the spatial join algorithms outlined in Figure 40. As mentioned in the previous section, Shadow Join and Clone Join are the key algorithms in this design space, and we now develop analytical cost models for these algorithms. The goal of analytical modeling is to understand the relative performance of the two algorithms, and to produce cost functions that can be plugged into a cost-based optimizer so it can choose the less expensive algorithm for a particular algorithm.

5.3.1 Notations

Let, \( R \) and \( S \) denote the two relations being joined. Let \(|R|\) denote the size of the relation \( R \) in pages, and let \(|R|\) denote the number of tuples in \( R \). Also, let \(|M|\) denote the size of main memory in pages. The system parameters for the model are shown in Table 11, and the parameters that characterize the data sets are shown in Table 12. The system parameters are measured in terms of the CPU cycles using a performance measuring tool quantify [Qua97]. In the cost formulae the CPU cost is calculated by multiplying the CPU cost in cycles with the CPU speed, which is measured in cycles per second and is set to 133 MHz.

The replication factor, \( F_{Repl} \), in Table 12 is the factor by which the cardinality of a declustered relation increases
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{I/O}$</td>
<td>Cost of a disk I/O</td>
<td>2 ms/page</td>
</tr>
<tr>
<td>$C_{Net}$</td>
<td>Cost of sending a page over the network</td>
<td>0.67 ms/page</td>
</tr>
<tr>
<td>$C_{FragBox}$</td>
<td>Cost of producing a partial spatial surrogate</td>
<td>900 cycles</td>
</tr>
<tr>
<td>$C_{Comp}$</td>
<td>Cost of comparing two integers</td>
<td>1 cycle</td>
</tr>
<tr>
<td>$C_{Hash}$</td>
<td>Cost of hashing on multiple attributes and inserting into a hash table</td>
<td>200 cycles</td>
</tr>
<tr>
<td>$C_{OIDLookup}$</td>
<td>Cost of looking up the node information from an OID</td>
<td>5 cycles</td>
</tr>
<tr>
<td>$C_{OIDComp}$</td>
<td>Cost of comparing (for sorting) two OIDs</td>
<td>20 cycles</td>
</tr>
<tr>
<td>$C_{EM}$</td>
<td>Cost of examining two spatial attributes for overlap or containment</td>
<td>20000 cycles</td>
</tr>
<tr>
<td>$S_{Page}$</td>
<td>Size of a page</td>
<td>8192 bytes</td>
</tr>
<tr>
<td>$S_{SpSur}$</td>
<td>Size of a partial spatial surrogate</td>
<td>48 bytes</td>
</tr>
<tr>
<td></td>
<td>CPU speed</td>
<td>133 MHz</td>
</tr>
</tbody>
</table>

Table 11: System Parameters.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_{Rep}$</td>
<td>Replication Factor</td>
<td>variable</td>
</tr>
<tr>
<td>$F_{Join}$</td>
<td>Join Selectivity</td>
<td>variable</td>
</tr>
<tr>
<td>$F_{Cand}$</td>
<td>Ratio of candidate list cardinality to final result cardinality</td>
<td>variable</td>
</tr>
<tr>
<td>$X_{Ovalp}$</td>
<td>Average number of rectangles that intersect a vertical plane sweep line</td>
<td>100 bytes</td>
</tr>
<tr>
<td>$S_{Tuple}$</td>
<td>Average size of a tuple</td>
<td>180 bytes</td>
</tr>
</tbody>
</table>

Table 12: Data Characteristics.

after a relation is declustered. This factor is always greater than 1. The join selectivity in Table 12 is the ratio of the cardinality of the output relation to the product of the cardinalities of the input relations.

### 5.3.2 Cost Formula for the Clone Join

Let us first derive the cost formula for the Clone Join (see Figure 42). The cost of redeclustering a relation $R$ based on the whole declustering strategy (D-W) is:

$$
C_{DW}(R) = |R| \times C_{I/O} \quad \text{(Cost of reading the input)}
+ \frac{|R| \times |R| \times C_{FragBox}}{2} \quad \text{(Cost of fragmenting tuples)}
+ |R| \times F_{Rep} \times C_{Net} \quad \text{(Send tuples to the next operator)}
$$

The next operator, the “local spatial join”, uses the PBSM algorithm [PD96]. This executes in two steps, the filter step and the refinement step. The filter step of the spatial join first extracts the MBR and the OID of the input tuples, and stores this information internally in a relation. Let $R_S$ and $S_S$ denote the internal relation that are created for the relations $R$ and $S$ respectively. The tuples in these relations have the same structure as a partial spatial surrogate. Next, the filter step joins the MBRs in these relations using a computational geometry plane-sweep algorithm. If the relations and are too large to fit in memory, then they are partitioned, and the partitions are joined (refer to Section 2.3 for the details). The number of partitions $P$ is given by:

$$
P = \frac{(|R| + |S|) \times S_{SpSur}}{S_{Page}}
$$

The total cost of the filter step of the spatial join of two relations $R$ and $S$ is given by:
\[ C_{\text{RedJoin}}(R, S) = (|R| + |S|) \times C_{I/O} \quad (\text{Cost of reading the inputs}) \]
\[ + |R| \times C_{\text{FragBox}} + |S| \times C_{\text{FragBox}} \quad (\text{Cost of extracting MBR}) \]
\[ + ([|R_S| + |S_S|) - \min(|R_S| + |S_S|, |M| - 2P)] \times 2 \times C_{I/O} \quad (\text{Cost of partitioning}) \]
\[ + |R| \times F_{\text{Repl}} \times \log(|R_P|) \times C_{\text{Comp}} \quad (\text{Cost of sorting rectangles}) \]
\[ + |S| \times F_{\text{Repl}} \times \log(|S_P|) \times C_{\text{Comp}} \quad (\text{Cost of sorting rectangles}) \]
\[ + (||R|| + ||S||) \times X_{\text{Optp}} \times C_{\text{Comp}} \quad (\text{Cost of the plane-sweep}) \]

where \(|R_P|\) is the number of tuples in a partition of \(R\) that is held in memory, and the size of the partial spatial surrogate relation, \(|R_S|\), is:
\[ |R_S| = \frac{|R| \times S_{\text{SpSur}} \times F_{\text{Repl}}}{S_{\text{Page}}} \]

The filter step produces an intermediate result consisting of OID pairs, with one of the OIDs in the pair pointing to a tuple in the relation \(R\) and the other pointing to a tuple in the relation \(S\). The refinement step fetches the tuples corresponding to these OIDs, and checks if the join predicate is actually satisfied. The refinement step first sorts the intermediate result on the OID pointing to the \(R\) tuple, and then (sequentially) fetches as many of the referenced \(R\) tuples as will fit in memory. Then, it walks through the set of \(R\) tuples in memory and fetches the \(S\) tuple corresponding to each \(R\) tuple. The two tuples are compared, and a result tuple is produced if the join predicate is satisfied. For further details refer to Section 2.3. The cost of this step is:
\[ C_{\text{Refinement}} = 2 \times |C| \times \min(|C|, |M| - \sqrt{|C|}) \times C_{I/O} \quad (I/O \text{ cost for sorting}) \]
\[ + |C| \times \log(|C|) \times C_{\text{OldComp}} \quad (CPU \text{ cost for sorting}) \]
\[ + |R| \times C_{I/O} \quad (\text{Read tuples of } R) \]
\[ + |S| \times \frac{|C| \times (S_{\text{Optp}} + S_{\text{SpSur}})}{S_{\text{Page}} \times |M|} \times C_{I/O} \quad (\text{Multiple passes for fetching S tuple}) \]
\[ + |C| \times C_{EM} \quad (\text{Evaluate the join predicate}) \]
\[ + F_{\text{Join}} \times (||R|| \times ||S||) \times F_{\text{Optp}} \times \frac{2 \times S_{\text{Optp}}}{S_{\text{Page}}} \times C_{\text{Net}} \quad (\text{Send tuples to the next operator}) \]

Here \(C\) is the intermediate result consisting of the list of OID pairs. The number of tuples in this relation is \(F_{\text{Join}} \times (||R|| \times ||S||) \times F_{\text{Optp}} \times F_{\text{CondList}}\), and the size of each tuple is the same as the size of an OID pair.

The last operator in the tree is a distinct operator that is used to eliminate duplicates that might be produced as a result of the replication used in the redeclustering step (refer to Figure 42). The distinct operator uses the an adaptive hashing algorithm [ZG90] to remove the duplicates. If required, the input is broken into partitions such that each partition fits in memory.

The input to the distinct operator is a relation \(T\), and the number of pages in this relation is given by the
formulae:

$$|T| = \frac{(|R| \times |S|) \times F_{Oult} \times F_{Join} \times 2 \times S_{tuple}}{S_{Page}}$$

The cost of the distinct operator is:

$$C_{Distinct} = 2 \times |T| - \min(|T|, |M| - \sqrt{|T|}) \times C_{I/O} \quad \text{(Partitioning cost)}$$

$$+ \quad (|R| \times |S|) \times F_{Oult} \times F_{Join} \times C_{Hash} \quad \text{(Cost of hashing)}$$

Thus the total cost of Clone Join is:

$$C_{Clone}(R, S) = C_{DW}(R) + C_{DW}(S) + C_{RedJoin}(R, S) + C_{Refinement} + C_{Distinct}$$

5.3.3 Cost Formula for the Shadow Join

We now derive the cost formula for the Shadow Join (Figure 41). The cost of redeclustering a relation R creating partial spatial surrogates (D-PSS) is:

$$C_{DPSS}(R) = |R| \times C_{I/O} \quad \text{(Cost of reading the input)}$$

$$+ \quad |R| \times C_{FragmentBox} \quad \text{(Cost of fragmenting tuples)}$$

$$+ \quad \frac{|R| \times S_{sp} \times S_{p}}{S_{Page}} \times F_{Repl} \times C_{Net} \quad \text{(Send tuples to next operator)}$$

The cost of joining the two sets of spatial surrogates is the same $C_{RedJoin}$ as in the previous section (without the cost of reading the inputs). This step produces a set of OID pairs. The cardinality of this set is $|R| \times |S|) \times F_{Join} \times F_{Repl} \times F_{Candidate}$, and the cost of redeclustering this set is:

$$C_{DOID}(R) = (|R| \times |S|) \times F_{Join} \times F_{Repl} \times F_{Candidate} \times F_{OidLookup}$$

$$+ \quad \frac{|R| \times |S| \times F_{Join} \times F_{Repl} \times F_{Candidate} \times F_{OidLookup}}{S_{Page}} \times C_{Net}$$

$$\quad \text{(Send tuples to the next operator)}$$

The last two operators in Figure 41 are similar as they both perform a join on OIDs. Each of these operators take as input a relation T. Every tuple in this relation has an OID attribute that is used in the join. The join operator "joins" by fetching the tuple corresponding to the OID. To avoid random I/Os, the join operator first sorts the relation T on the OID attribute, and then fetches the tuples referred by the OIDs. The cost of sorting this relation T is:

$$C_{OidSort}(T) = 2 \times |T| - \min(|T|, |M| - \sqrt{|T|}) \times C_{I/O}$$

$$\quad \text{(I/O cost for sorting)}$$

$$+ \quad |T| \times \log(|T|) \times C_{OidComp} \quad \text{(CPU cost for sorting)}$$

The cost of reading the T tuples after sorting is:

$$C_{Fetch} = \frac{|T| \times S_{tuple}}{S_{Page} \times F_{Duplicates}} \times C_{I/O}$$
In this equation, \( F_{Duplicates} \) is the average number of duplicate entries in the relation \( T \). In the first OID join operator (operator 4 in Figure 41) this factor is equal to \( F_{Repl} \), and in the second OID join operator (operator 5 in Figure 41) this factor is equal to 1. The number of tuples in the relation \( T \) is \( ||R|| \times ||S|| \times F_{Join} \times F_{Repl} \times F_{card} \) in the first join operator, and \( (||R|| \times ||S||) \times F_{Join} \times F_{card} \) in the second join operator. The average size of a \( T \) tuple is \( 2 \times S_{SpSort} \) in the first OID join operator, and \( S_{SpSort} + S_{Tuple} \) in the second OID join operator. Finally, we have to add the cost of the exact match step (this happens in the second OID join operator). This cost is simply: \( C_{RefinementPSS} = ||T|| \times C_{EM} \)

Thus, the total cost of Shadow Join is:

\[
C_{Shadow}(R,S) = C_{DPSS}(R) + C_{DPSS}(S) + C_{ReclJoin}(R,S) + C_{OID}(R) + C_{OiSort}(T^1) + C_{Fetch}(T^1) + C_{OiSort}(T^2) + C_{Fetch}(T^2) + C_{RefinementPSS}
\]

where \( T^1 \) and \( T^2 \) denote the two "T" relations mentioned above.

### 5.4 Analytical Predictions

We now use the equations developed in the previous section to predict the behavior of the Clone and the Shadow join algorithms. We explore the effects of the following three parameters:

1. Join Selectivity
2. Replication Probability
3. Fertility Ratio

Join selectivity is the ratio of the cardinality of the output relation to the product of the cardinalities of the input relations. Replication probability is the probability that a tuple in an input relation will get replicated when it is declustered. Referring to Table 12, the replication probability is the same as \( F_{Repl} - 1 \). Both the Clone join and the Shadow join use the MBRs of the tuples during the join. The result of joining the MBRs is an approximate answer set that is called the candidate set. In the Clone Join (Figure 42), the candidate set is produced during the local spatial join (as explained in Section 5.3.2, this step is executed right after the filter step of the local join). In Shadow Join (Figure 41), the candidate set is produced at the end of the local spatial join operator (operator 3). The ratio of the cardinality of the final result set to the cardinality of the candidate set is called the **fertility ratio**. It represents how “successful” the candidate set is in producing the final result set. A low fertility ratio implies that many of the tuples in the candidate set do not satisfy the final join predicate.

In the following analytical experiments, we vary each of the parameters above one at a time. The default values for these parameters are: replication probability = 0.07, fertility ratio = 0.2, and join selectivity = 0.5\( e^{-6} \). These values were chosen based on actual experiments with the DCW [DCW92] data sets. The cardinality of each input relation is set to 1 million.
5.4.1 Effect of Join Selectivity

![Effect of Join Selectivity](image)

Figure 43: Analytical Modeling: Effect of Join Selectivity (\(||R|| = ||S|| = 1M\), replication probability = 0.07, fertility ratio = 0.2.

Figure 43 shows the effect of join selectivity on the two algorithms. As shown in the figure, when the join selectivity is small (compare Shadow-0.1e-6 with Clone-0.1e-6), Shadow join outperforms Clone join. As the join selectivity increases (compare Shadow-1e-6 with Clone-1e-6), Clone join performs better than Shadow join. Shadow join has to move candidate tuples between operators 3 and 4, and between operators 4 and 5 (Figure 41). Many of these candidate sets are eventually not part of the final result set. As the join selectivity increases, the size of the candidate sets that are passed around also increases, thereby, causing Shadow join to perform poorly.

5.4.2 Effect of Replication Probability

Figure 44 shows the effect of the replication probability on the two algorithms. The predicted performance of Shadow and Clone join is plotted for replication probabilities of 2% and 40%. Increasing the replication probability has little effect on the performance of Shadow join. This effect is because Shadow join uses partial spatial surrogates that are designed to reduce the overhead associated with replication (recall the results plotted in Figure 38). Clone Join, on the other hand replicates entire tuples in the partitioning phase, and hence sees a rapid degradation in performance as the replication overhead is increased.

5.4.3 Effect of Fertility Ratio

Figure 45 shows the effect of the fertility ratio on the performance of the two join algorithms. For low fertility ratios, Clone Join outperforms Shadow Join (compare Clone-0.1 with Shadow-0.1). As the fertility ratio increases
Figure 44: Analytical Modeling: Effect of Replication Probability (\(||R|| = ||S|| = 1M\), join selectivity = 0.5e-6, fertility ratio = 0.2).

Figure 45: Analytical Modeling: Effect of Fertility Ratio (\(||R|| = ||S|| = 1M\), join selectivity = 0.5e-6, replication probability = 0.07)
(compare Clone-1 with Shadow-1), the relative performance of the two algorithms is reversed. When the fertility ratio is low, the cardinality of the candidate set is large compared to the final result. Shadow Join has to move this candidate set, in one form or another, a couple of times: once between the local join and the first OID join, and then between the two OID joins. Clone join on the other hand, prunes the candidate set locally (in the local join operator in Figure 42), and only transfers the pruned set once to the distinct operator. Consequently, lower fertility ratios favor the Clone join algorithm.

5.4.4 Summary of the Analytical Performance Comparisons

To summarize, one difference between Shadow join and Clone join is that Shadow join uses partial spatial surrogates for replication. Partial spatial surrogates have a very low replication overhead, and, consequently, Shadow join is rather immune to changes in the replication characteristics of the data. Clone join, on the other hand, replicates entire tuples, and its performance degrades rapidly if the underlying data has a high probability of requiring replication.

Another difference between the two algorithms is the number and size of the intermediate result sets that are generated during the execution of the algorithms. Clone Join has one intermediate result set (see Figure 42) that is used to transfer data between the local spatial join and the distinct operators. The size of this intermediate result is largely influenced by the join selectivity and the replication probability. Shadow join, on the other hand, generates two intermediate result sets. The first intermediate result set is produced by the local spatial join and sent to the first OID join (operators 3 and 4 respectively). The second intermediate result set is produced by the first OID join operator (operator 4) and sent to the second OID join operator (operator 5). The sizes of these intermediate result sets are largely influenced by the fertility ratio and the join selectivity. A lower fertility ratio, or a larger join selectivity implies larger intermediate result sets. Consequently, a low fertility ratio or a high join selectivity has a negative effect on Shadow Join’s performance, allowing Clone Join to outperform it under these conditions.

5.5 Experimental Performance Comparison

In this section, we compare the performance of Shadow Join and Clone Join based on actual implementation of these algorithms. These algorithms were implemented inside Paradise, a scalable geo-spatial database system [PYK+97].

5.5.1 Data Sets

The data sets that are used in this benchmark come from the DCW data product [DCW92]. We used the drainage, road and rail data sets. The drainage data set describes, using polylines, drainage features, such rivers, streams, canals, etc., for the entire world. Similarly, the road and the rail data set describe, using polylines, roads and railway lines for the entire world. The characteristics of this data set are summarized in Table 13.
<table>
<thead>
<tr>
<th></th>
<th>Tuple Count</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drainage</td>
<td>1.73 M</td>
<td>300 MB</td>
</tr>
<tr>
<td>Road</td>
<td>0.7 M</td>
<td>100 MB</td>
</tr>
<tr>
<td>Rail</td>
<td>0.14 M</td>
<td>18 MB</td>
</tr>
</tbody>
</table>

Table 13: DCW Data Set.

5.5.2 Testbed Description and DBMS Configuration

For the tests conducted in this paper we used a cluster of 17 Intel eXpress PCs each configured with dual 133 Mhz Pentium processors, 128 Mbytes of memory, dual Fast & Wide SCSI-2 adapters (Adaptec 7870P), and 6 Seagate Barracuda 2.1 Gbyte disk drives (ST32500WC). Solaris 2.5 was used as the operating system. The processors are connected using 100 Mbit/second Ethernet and a Cisco Catalyst 5000 switch that has an internal bandwidth of 1.2 Gbits/second. Five of the six disks were configured as "raw" disk drives (i.e. without a Solaris file system). Four were used for holding the database and the fifth for holding the log. The sixth disk was initialized with a Solaris file system. This disk was used for holding system software as well as swap space. The four disk drives used to hold the database were distributed across the two SCSI chains.

Paradise was configured to use a 32 MByte buffer pool. Although this is a small buffer pool relative to the 128 MByte of physical memory available, Paradise does much of its query processing outside the buffer pool in dynamically allocated memory. The maximum process size we observed during benchmark execution was about 90 Mbytes. Thus, no swapping occurred. In all test configurations, all relations were partitioned across all the database storage disks (4 per processor) in the system.

5.5.3 Validation of the Analytical Model

We tried validating the analytical model that was presented Section 5.3 with the implementation of the algorithms on the hardware platform just described. However, we found that while the analytical model and the implementation did agree on the general trends and the relative behavior of the algorithms, the actual numbers were off by as much as 50%. This anomalous behavior is the result of the analytical cost model being very simple and not accounting for many of the costs that an actual implementation has. An actual implementation has contention for resources such as the network, disk and memory, and there are many CPU costs for creating and deleting tuples, etc. None of these are accounted for in the simple analytical cost models. As the following experiments show, the predications made by the analytical model are validated by the actual implementation.

5.5.4 Experimental Results

Experiment 1

Figure 46 shows the speedup of the two algorithms while joining the Drainage relation and the Road relation. In a parallel database system, speedup is defined as the ability of the system to perform a task in half the elapsed time if the system has twice as much hardware [DG99].
For this experiment, the result relation has 0.7 M tuples and is 300 MB in size. The data characteristics are: Replication Probability = 0.07, Fertility Rate = 0.20, and Join Selectivity = 0.53e – 6. As shown in the figure, both algorithms have close to linear speedup. As the number of nodes in the system doubles, the query execution time reduces approximately by half.

For these parameters, both the algorithms have comparable performance.

![Graph](image1.png)

**Figure 46:** Join Drainage and Roads (Replication Probability = 0.07, Fertility Rate = 0.20, Join Selectivity = 0.53e – 6)

![Graph](image2.png)

**Figure 47:** Effect of Join Selectivity: Join Drainage and Roads (Replication Probability = 0.07, Fertility Rate = 0.20, **Join Selectivity** = 0.27e – 6)

**Experiment 2: Effect of Join Selectivity**

Next, we examine the effect of join selectivity on the two algorithms. To study this case, we needed a data set which has a join selectivity different from the join selectivity in Experiment 1. In trying to set up such an experiment, we first considered joining the Drainage and Rail tables, or the Rail and Road tables. Unfortunately, both these joins have the same characteristics as the join described in Experiment 1. Hence, to study the effect of join selectivity, we again joined the Drainage and Road tables, but arbitrarily dropped half of the tuples in the intermediate result tables (the candidate set) that are produced during the execution of both the algorithms. This process of reducing the cardinality of the candidate set by half, has the effect of approximately halving the cardinality of the final result set. For the Shadow join (refer to Figure 41) every second tuple that is produced by the local join operator (operator 3) is dropped on the floor instead of sending it to the next operator. For the Clone join (refer to Figure 42), every second tuple that is produced by the filter step of the local spatial join is dropped. The result of executing this experimental setup is plotted in Figure 47. As predicted by the analytical model, the lower join selectivity favors Shadow Join (see Section 5.4.1 for an explanation).
Experiment 3: Effect of Fertility Ratio

We now explore the effects of the fertility rate on the two algorithms. The data set for this experiment is produced using a technique similar to that used in Experiment 2. To decrease the fertility rate, we double the number of tuples in the intermediate results. For the Shadow join (refer to Figure 41) we produce two tuples for every tuple that is produced by the local join operator (operator 3). For the Clone join (refer to Figure 42), every tuple that is produced by the filter step of the local spatial join is added to the intermediate result twice. Effectively, this technique doubles the size of the candidate lists while maintaining the size of the final result set. The result of running this experiment is shown in Figure 48. Again, as predicted by the analytical model, a lower fertility ratio favors the Clone Join algorithm (see Section 5.4.3 for an explanation).

Figure 48: Effect of Fertility Ratio: Join Drainage and Roads (Replication Probability = 0.07, Fertility Rate = 0.11, Join Selectivity = 0.53ε – 6)

Figure 49: “Stitched” Data, Effect of Replication: Join Stitched-Road and Stitched-Rails (Replication Probability = 0.18, Fertility Rate = 0.23, Join Selectivity = 1.3ε – 6)

Experiment 4: Effect of Replication Probability

Next, we examine the effect of replication on the two join algorithms. For this experiment, we took the DCW data and “stitched” it. In the original DCW data set, a single feature, such as a road, is broken into a bunch of smaller road segments. Each segment is then stored in the database as a separate tuple. This fragmentization is done because road segments often have associated information such as the zip code of the area that they are passing through. When this zip code information changes for a road, a new road segment is produced.

We produced a data set by “stitching” the spatial features (and dropping the associated information such as zip codes). This data set has features that span larger areas of space, and hence is more likely to require replication when declustered. The characteristics of this data set are summarized in Table 14.

From Table 14, we observe that the stitching process dramatically changes the characteristics of the road and the rail relations. We use these two relations to evaluate the effect of high replication probability on the performance
of the two join algorithms. Figure 49 shows the results of this experiment. When the replication probability is high, Shadow join outperforms Clone join (again, for the same reasons described in Section 5.4.2).

**Experiment 5: Multiple Finer Approximations**

Both the Shadow Join and Clone join algorithms use an approximation of the spatial attribute to perform a “preliminary” join. In both algorithms, the approximation is a rectangle. Clone join uses the minimum bounding rectangle as the approximation (this is done in the filter step of the local spatial join), whereas Shadow join uses partial spatial surrogates as the approximations. These approximations are conservative profiles of the spatial attribute, and are used because joining them is more efficient then joining the actual spatial attributes and because they take fewer bytes to represent in memory (allowing us to fit larger data sets in memory). The join using this approximation quickly produces a candidate answer set that can then be pruned by examining the actual spatial attributes. However, an approximation like the minimum bounding rectangle (MBR) can be a poor representation of the spatial attribute. As an example, consider the river shown in Figure 50. The MBR approximating this river has a lot of dead space — space that is not actually covered by the spatial attribute. Now consider joining this river with the road shown in Figure 50. The road overlaps the MBR of the river, but does not intersect the river at any point. However, the join using the MBR will add this river-road pair for consideration of membership in the final result set. Another way of approximating the same river attribute is shown in Figure 51. Here, instead of approximating the river by one rectangle (the MBR), it is approximated by five smaller rectangles. These rectangles, called fragment boxes, collectively approximate the river tuple, but are a better approximation of the river than the MBR. The partial spatial surrogates create similar fragment boxes when a spatial attribute overlaps tiles that are mapped to different nodes. However, it is interesting to consider the effect of forcing the use of these fragment boxes, beyond what is automatically imposed by the declustering mechanism.

A similar problem occurs in centralized spatial join algorithms, since many centralized spatial join algorithms first join using the minimum bounding rectangle of the spatial attribute. It is shown in [BKSS94] that approximating the spatial attributes by a closer approximation (like the multiple fragment boxes here) enhances the performance of the centralized spatial join algorithm. We now perform a simple experiment to examine the effect of multiple finer approximations on the Shadow join and the Clone join. For this experiment, whenever the area of the MBR of a spatial attribute is greater than a certain threshold, it is approximated by a number of smaller rectangles. This threshold is arbitrarily set to 0.01% of the area of the universe. Also, rather simplistically, the number of the approximations used to represent such large attributes is set to 10 times the ratio of the MBR area to the threshold area. Thus, a spatial attribute whose MBR has an area two times over the threshold limit is approximated by 20 small rectangles.
Table 15 shows the results of joining the Drainage relation and the Road relation (the same data set as in Experiment 1). From this table, we observe that the multiple approximation did not help, but it did not hurt either. The benefit of a better approximation is offset by the cost of actually forming these approximations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clone</td>
<td>1105.9</td>
<td>586.2</td>
<td>287.8</td>
</tr>
<tr>
<td>Clone-Multiple-Approximations</td>
<td>1104.0</td>
<td>583.7</td>
<td>295.0</td>
</tr>
<tr>
<td>Shadow</td>
<td>1080.9</td>
<td>596.0</td>
<td>308.3</td>
</tr>
<tr>
<td>Shadow-Multiple-Approximations</td>
<td>1072.7</td>
<td>585.3</td>
<td>306.0</td>
</tr>
</tbody>
</table>

Table 15: Effect of Multiple Finer Approximations on Join Drainage and Road

Table 16 shows the results of joining the stitched Road relation and the stitched Rail relation (the same data set as in Experiment 4). With the stitched data set, the spatial features are longer, and hence better approximations are more crucial. The table shows that using multiple approximations improves the performance of both the algorithms by 20 to 35%. Shadow join benefits a little more than Clone join because better approximations increases the fertility ratio, and this has a bigger impact on Shadow join (refer to the discussion in Experiment 3, Section 5.5.4).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clone</td>
<td>686.1</td>
<td>451.1</td>
<td>249.1</td>
</tr>
<tr>
<td>Clone-Multiple-Approximations</td>
<td>576.0</td>
<td>337.0</td>
<td>201.8</td>
</tr>
<tr>
<td>Shadow</td>
<td>553.3</td>
<td>289.2</td>
<td>167.4</td>
</tr>
<tr>
<td>Shadow-Multiple-Approximations</td>
<td>425.2</td>
<td>213.2</td>
<td>128.9</td>
</tr>
</tbody>
</table>

Table 16: Effect of Multiple Finer Approximations on Join Stitched Road and Rail
5.6 Conclusions

Based on the declustering strategies proposed in the previous chapter, this chapter explores the design space of various parallel spatial join algorithms. The design space contains two key algorithms — the Clone Join and the Shadow Join. Both analytical cost modeling and actual implementations show that both these algorithms exhibit good speedup characteristics. The relative performance of the two algorithms depends on the data characteristics and the selectivity of the join.
Chapter 6

Parallel Spatial Aggregates

In Chapter 3 spatial aggregates were introduced, and a simple algorithm for evaluating a spatial aggregate in a centralized system was presented. This chapter explores parallel algorithms for evaluating the spatial aggregate operation.

6.1 Parallel Spatial Aggregate Algorithm

6.1.1 Naive Evaluation

Let us begin by examining the problems associated with parallelizing the spatial aggregate algorithm proposed in Chapter 3. Consider the following two relations

**River** (name: **String**, shape: **Polyline**, category: **Integer**)

**Cities** (name: **String**, population: **Integer**, location: **type Point**)

Now consider the evaluation of the query: Find the closest river to every city with a population greater than one million. In a centralized system, this query can be evaluated using the operator tree shown in Figure 26 (in Chapter 3). In this operator tree, a select operator on the city relation first selects all the cities that have a population greater than one million, and these tuples are then fed to a “join” operator. For each city tuple, the join operator repeatedly probes the spatial index on River.shape with a circle till it finds the closest river tuple.

Now, consider a parallel system with two nodes. The operator tree for a naive parallel evaluation of the aggregate query is shown in Figure 52. In this naive approach, a copy of the centralized operator tree is replicated at every node. In this operator tree, the select operator replicates the selected city tuples, and sends them to every “join with aggregate” operator running in the system. The “join with aggregate” operator looks at its local portion of the river relation, and finds the locally closest river. At the end of the “join with aggregate” operation, every node in the system produces the locally closest river for every city tuple that has been selected. The final global aggregate operator collects these partial results from all the nodes, and locates the closest river to each city tuple. The actual instances of the operators running on the two nodes is shown in Figure 53. In this figure, there is only one instance of the global aggregate operator in the system. If very few city tuples get selected, thereby implying only a few groups, then there is little parallelism in the global aggregation phase. Starting only one instance of the global aggregate operator might be more efficient than starting an instance of the global aggregate at every node in the the system [Sha96].
Figure 52: Parallel Aggregate Evaluation, Naive Approach: Operator Tree

Figure 53: Parallel Aggregate Evaluation, Naive Approach: Operator Tree Instance
6.1.2 Spatial Semi-Join

The approach above benefits only marginally from parallelism. Every node evaluates the closest aggregate for every selected city tuple. The (small) benefit of using parallelism is that every node examines only the portion of the river relation that is local to its node. To make more effective use of parallelism, the following approach can be used:

1. Redistribute the river relation on the location attribute using full replication (the D-W policy proposed in Section 4.3.1). This step can be skipped if the river relation is already declustered in this way.

2. Redistribute the city relation on the location attribute using the same deaggregation policy as used in the previous step. This step can also be skipped if the city relation is already deaggregated in this way.

3. If a spatial index does not exist on the location attribute of the river relation, then build one on the fly. (The index is local to each node, and is built only on the fragment of the river relation at that node).

4. Finally, execute the operator tree shown in Figure 54.

The spatial semi-join looks at the city tuple, and forms the largest circle that is completely contained in the tile corresponding to that node (as explained in Section 4.3, tiling is used in the deaggregation function). The center of the circle is the location of the city (a point feature). The semi-join operator then uses this circle to probe the index on the river. If any tuple falls within this circle, then the closest drainage feature has to be one of these features, in which case, the city tuple is sent to the “join with aggregate” operator on the same node. If the index probe does not return any tuples, then the closest feature could be on any node in the system, and the city tuple is replicated and sent to all the nodes in the system. A example of the execution of the spatial semi-join
is shown in Figure 55. The figure shows the processing of a city tuple labeled Madison. Since there are no river tuples within the largest circle that is centered around Madison and contained within the tile to which Madison belongs, the Madison tuple is replicated and sent to all the other nodes in the system. Every node then looks at its local set of rivers and finds the locally closest river to Madison. Finally, the global aggregate collects the closest river feature to a city from every node, and produces the final result.

### 6.2 Performance Evaluation

This section contains the results of implementing both the parallel spatial aggregate algorithm described in the previous section in Paradise. The data set used in this study is the drainage and the cities data set from DCW [DCW92]. The definition of these two relations in Paradise is as given below.

```plaintext
Cities (id String,               // unique feature identifier
    containing_face String,  // feature-id of the
    type Integer,            // category of the city
    location Point,         // coordinates of the city
    name String);           // name of the city

Drainage (id String,            // unique feature identifier
    type Integer,           // drainage feature type
    shape Polyline);        // shape of the feature
```

The characteristics of the data is shown in Table 17.

The testbed and DBMS configuration for this performance was the same as described in Section 5.5.2. For this experiment, the spatial declustering function used 10,000 tiles.

The following query was run on this data set.
<table>
<thead>
<tr>
<th></th>
<th>Tuple Count</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drainage</td>
<td>1.73 M</td>
<td>300 MB</td>
</tr>
<tr>
<td>Cities</td>
<td>0.25 M</td>
<td>20 MB</td>
</tr>
</tbody>
</table>

Table 17: DCW Cities and Drainage Data Set.

```sql
select closest(drainage.shape, cities.location),
cities.location
from drainage, cities
where cities.location overlaps drainage.shape
  and cities.type = LARGE_CITY
group by cities.location
```

This query finds the closest drainage feature to every large city. In this data set, 20040 cities satisfy the selection criteria for large cities. The speedup results of running this query on the same testbed described in Section 5.5.2 is shown in Figure 56. This graph shows the results of running the query with and without the semi-join operator (the operator trees shown in Figure 52 and 54 respectively). The figure shows that without the semi-join operator, the performance degrades as the number of nodes in the system is increased! Parallelism does not help the case when there is no semi-join, since every node processes all the selected city tuples. On the contrary, parallelism has an additional cost. Without the semi-join, the algorithm replicates every selected city tuple and sends these to all the nodes. This replication overhead increases quadratically with the number of nodes, causing a degradation in performance. The figure also shows that using the semi-join operator gives better performance. However, the speedup obtained by using the semi-join is still sub-linear. As the number of nodes increases, the number of cities that do not find a local drainage feature during the spatial semi-join increases. Since these city tuples have to be replicated, we see less than linear speedup. The figure also shows that in the 4-node configuration using the semi-join actually causes a small degradation in performance. This degradation of performance in the 4-node configuration is because in this configuration the overhead associated with fully replicating the selected city tuples is small. The semi-join has an associated overhead (since it probes the spatial index), and with a small number of nodes this overhead outweighs the benefit of not having to process every selected city tuple at every node.

### 6.3 Conclusions

In this chapter, an algorithm for evaluating the spatial aggregate in a parallel environment has been presented. The algorithm makes use of a spatial semi-join operator that allows each node to process a substantial portion of the aggregate locally. Experimental results show that parallelism can be exploited, though to a limited extent, using the spatial semi-join operator. The intrinsic nature of spatial data makes it hard to divide the aggregation work into independent tasks that can be processed in parallel, and, as a result, the speedup of the parallel aggregate algorithm is sub-linear.
Figure 56: Find the Closest River to Every Large City
Chapter 7

Conclusions

7.1 Summary

There are two emerging trends with geo-spatial applications. Application are demanding more complex query processing on ever increasing volumes of spatial data. To support these application, a spatial database system must have efficient data storage and query processing algorithms. This thesis has investigated efficient algorithms for two crucial spatial operations: spatial join and spatial aggregate. Both these operations are expensive to compute, and hence efficient algorithms for these are crucial to the overall performance of a spatial database system. The initial chapters of the thesis proposed new centralized algorithms for spatial join and spatial aggregate operations. In Chapter 2, a new spatial join algorithm, called Partition Based Spatial-Merge (PBSM), was proposed and its performance was compared with existing centralized spatial join algorithms. PBSM was shown to be very effective for evaluating “ad-hoc” spatial joins, when none of the input data sets to the join have an pre-existing spatial indices built on them. Chapter 3 proposed a spatial aggregate algorithm that uses a spatial index to speed up the evaluation. It was shown that this algorithm can dramatically reduce the time it takes to evaluate the spatial aggregate operation. Furthermore, this algorithm can be easily implemented with the basic spatial index primitives that spatial indices already provide.

The second half of the thesis explored possible ways of using parallelism for storing and querying spatial data. Chapter 4 explored two strategies for declustering spatial data in a parallel shared-nothing spatial database system. Because of the space-spanning property of spatial data, both these declustering strategies require some form of replication. Chapter 5 examined parallel algorithms for evaluating a spatial join operation. The design space of parallel spatial join algorithms was mapped out, and two algorithms were identified as the primary algorithms in the design space. These two algorithms were compared using analytical cost model and actual implementations. It was shown that these algorithms can effectively exploit parallelism. Parallel spatial aggregates was the focus of Chapter 6, where a parallel spatial aggregate algorithms was proposed. Results from actual implementations were less optimistic in this case. While there is some gain from using parallelism, the speedup obtained is sub-linear. The intrinsic nature of spatial data make is really hard to divide the aggregation work into sub-tasks that can be evaluated in parallel. Sub-linear speedup results because the parallel algorithm has to duplicate a part of the task, and has to run another small task in serial mode.

In conclusions, efficient algorithms for evaluating the spatial join and the spatial aggregate operation dramatically improve the performance of a spatial database system, and these must be incorporated in any spatial database system that aims to provide fast responses to user queries. For large problem sizes, a spatial database system can effectively exploit parallelism. Parallelism is very effective for evaluating the spatial join operation, and linear
speedup is achievable. However, with the spatial aggregate linear speedup is not always possible. Nevertheless, the spatial aggregate does benefit significantly from using parallelism.

7.2 Future Work

The declustering techniques for parallel databases that are explored in this thesis use static partitioning of the space. However, there are some proposals for using a dynamic space partitioning techniques. For example, one could use the leaf nodes of a global R-tree to partition the space into overlapping regions, and spread the leaf pages to various nodes in the parallel system. Dynamic space partitioning schemes adapt to the underlying data easily, but can be expensive to maintain; the leaf of the global R-tree can be a potential bottleneck, and splits of leaf pages require specialized placement techniques. It would be interesting to examine the affect of dynamic space partitioning schemes on parallel spatial join algorithms.

This thesis has focussed on efficiently storing and processing spatial data. A simple relational tuple can be viewed as a point in a multi-dimensional space. Data mining application frequently perform a multi-attribute join on these “multi-dimensional” point tuples. Some of the spatial join techniques might find application in this area too.
Bibliography


[Rot91] D. Rotem. “Spatial Join Indices”. In IEEE Transactions on Knowledge and Data Engineering, Kobe, April 1991.


