Clone Join and Shadow Join: Two Parallel Spatial Join Algorithms*

Jignesh M. Patel†
EECS Department
University of Michigan
Ann Arbor, MI 48109-2122
jignesh@umich.edu

David J. DeWitt
Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
dewitt@cs.wisc.edu

ABSTRACT

Spatial applications frequently need to join two data sets based on some spatial relationship between objects in the two data sets. This operation, called a spatial join, is an expensive operation and in the past many algorithms have been proposed for evaluating the spatial join operation on a single processor system. However, the use of parallelism for handling queries involving large volumes of spatial data has received little attention. In this paper, we explore the use of parallelism for evaluating the spatial join operation. We first propose two strategies for storing spatial data in a parallel database system. We propose a number of spatial join algorithms based on these declustering strategies. Two algorithms are identified as the key algorithms in this design space. We analyze these two algorithms both analytically and experimentally. The experimental evaluation uses real data sets and is based on an actual implementation in a parallel database system. The experiments show that both algorithms can effectively exploit parallelism.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*

General Terms

Algorithms, Design, Experimentation, Performance.

Keywords

Spatial Join, Spatial Declustering, GIS.

*Funding for this research was provided by NASA under contracts #USRA-5555-17, #NAGW-3895, and #NAGW-4229 and ARPA through ARPA order number 017 monitored by the by the U.S. Army Research Laboratory under contract DAAB07-92-C-Q508, and by a gift donation from NCR Corp. to the University of Michigan.

†Part of this work was done while the author was at the University of Wisconsin.

ACM GIS 2000, Washington D.C, USA

1. INTRODUCTION

Spatial applications frequently need to combine two data sets based on some spatial relationship between objects in the two data sets. For example, given a polygon data set representing corn fields and another polygon data set representing areas affected by soil erosion, a soil scientist studying the effect of soil erosion on corn fields may ask the system to find corn fields that overlap with soil erosion polygons. This operation of combining two spatial data sets is called a spatial join. A spatial join operation, just like its counterpart in the relational world, is expensive to compute and very resource intensive. Recognizing the impact that a good spatial join algorithm can have on the overall performance and usability of a spatial database system, many algorithms for evaluating the spatial join operation in a centralized system have been proposed [18, 2, 6, 11, 7, 15, 10, 16, 19]. However, joining very large spatial data sets has received little attention. Large volumes of spatial data are now available from a variety of sources. 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 1:100,000 resolution is 19GB; such data sets for the entire world, and at higher resolutions will be much larger. In the near future, we will see even larger data sets. High resolution satellite images are now becoming available for commercial purposes at a very economical price [23]. A variety of applications that use these high resolution satellite images are now starting to emerge. Feature extraction algorithms can be used to identify features in an image, and these features can be stored in a database system as polygons, points, polylines, etc. Then, the applications can pose queries on these features. This feature extraction process can easily generate very large spatial data sets. With this inevitable increase in the volume of spatial data, there is clearly a need for efficient spatial join algorithms that operate on large data sets. In the relational world, as the sizes of databases increased, using parallelism to store and query large data sets was very effective. It is natural then to attempt to use parallelism for storing and querying large spatial data sets. This paper explores various algorithms for evaluating the spatial join operation in a parallel spatial database system. First, strategies for declustering spatial data are examined. Spatial data can be declustered using either a static or a dynamic partitioning scheme. Static partitioning divides the underlying space into regions, and maps the regions to nodes in the parallel system. Dynamic partitioning inserts the spa-

tial objects into a spatial index, like an R-tree [8], and maps the leaf nodes of the R-tree to nodes in the parallel system. In [26] it was shown that for spatial joins static partitioning is superior to dynamic partitioning. Consequently, this paper only focuses on declustering strategies that are based on static partitioning of the underlying space. First, this paper proposes two different declustering strategies. Then, based on these two declustering strategies, the design space for parallel spatial join algorithms is explored. Two algorithms, Clone Join and Shadow Join are identified as the key algorithms in this design space. An analytical model is used to investigate the characteristics of these two algorithms. These two algorithms are also implemented in Paradise, a parallel spatial database system, and this paper also evaluates these two algorithms using this implementation and real data sets. The experiments show that both these algorithms exploit parallelism effectively.

2. RELATED WORK

In this section we review the related work in the area of parallel spatial database systems. The paper by Tan and Yu [24] proposes techniques for declustering spatial data, and evaluates the effect of these declustering techniques on spatial selections. The evaluation uses synthetically generated uniformly distributed data, and does not consider data distribution skew. Another paper by Abel et al. [1] 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 [12] explore the use of parallelism to accelerate the performance of spatial selections. They examine various placement policies for distributing the leaves of an R-tree [8] across multiple disks in a system. The focus of the paper is limited to spatial selections, and the target parallel architecture is a single processor with multiple disks attached to it. This idea was later extended by Koudas et al. [14] to decluster spatial data on a shared-nothing architecture. Again, the scope of the study is limited to queries with spatial selections. Parallel range selection algorithms and dynamic data partitioning strategies has also been examined by Shekar et al. [22]. Hoel and Samet [9] have examined the use of PMR Quadtrees, R+-trees, and R-trees for evaluating the spatial join on a Thinking Machines, CM-5 platform. No I/O is considered in the paper, as all the data is always resident in main memory. Similar tree-based join algorithms have been proposed by Brinkhoff et al. [3] using two R*-trees for performing a spatial join in a shared-disk environment.

Recently, Zhou et al. [26], 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. Using the spatial partitioning function proposed in [19], the data is distributed to partitions by first dividing the space into cells. The cells are then mapped to partitions. As in [19] 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 [13, 25], 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.

3. SPATIAL DECLUSTERING

In a parallel shared-nothing system, the main source of parallelism is partitioned parallelism [5]. 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 parallelize spatial operations like the spatial join, one must first explore various declustering techniques that can be used to distribute spatial data across nodes in a parallel database system. In this section, we explore two such declustering techniques.

3.1 Declustering Using Replication

In this declustering strategy, the universe of the spatial attribute is divided into a number of tiles. (The universe of a particular spatial attribute is defined as the minimum rectangle that covers that spatial attribute for all the tuples in the relation.) The number of tiles is chosen to be much larger than the number of nodes in the system. Each tile is assigned a number, and a hash function is used to map the tile number to a node. A spatial object that is entirely contained within a tile, is assigned to the node corresponding to that tile. Spatial objects that overlap multiple tiles are replicated in all the nodes that correspond to the tiles that the object overlaps. The use of tiling reduced the effects of skewed data distribution, which can cause severe performance problems in a parallel database system [25]. Figure 1 illustrates a tiling scheme in which the universe is partitioned into 16 tiles that are mapped to 4 nodes. Note that this declustering strategy is similar to the spatial partitioning function that is used internally in the PBSM spatial join algorithm [19].

For the remainder of this paper, this declustering strategy is referred to as **D-W** ("decluster using whole tuple replication").

3.2 Partial Spatial Surrogate

The previous declustering strategy can have a very high replication overhead, especially when many spatial objects overlap with multiple tiles (see [20] for a detailed experimental analysis quantifying this overhead). To mitigate the 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 object identifier (OID) of the tuple and the

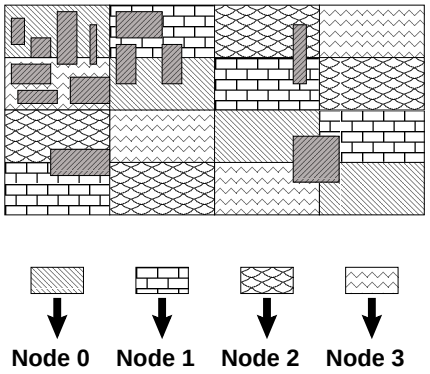

Figure 1: Declustering Using Tiling and Replication

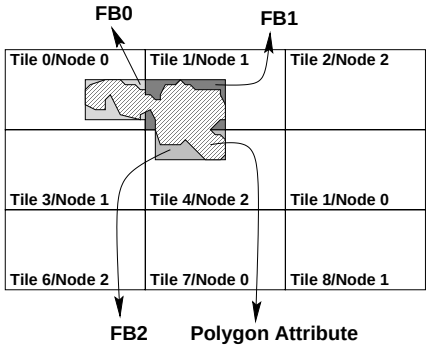

Figure 2: Declustering Using Spatial Surrogates

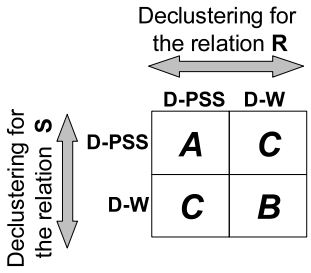

Figure 3: Design Space of Spatial Join Algorithms

minimum bounding rectangle (MBR) of the part of the spatial attribute that overlaps the tile 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 2 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.

For the remainder of this paper, this declustering strategy is referred to as **D-PSS** ("decluster creating partial spatial surrogates").

Both these declustering strategies have been evaluated and analyzed using a variety of data sets in [20].

4. PARALLEL SPATIAL JOINS

Having discussed declustering techniques, this section now examines various parallel algorithms for the spatial join operation. 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 redeclustered on their (spatial) join attributes. Before the join, if one or both the 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 D-W and the other relation could be declustered using 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 centralized spatial join algorithm. In this paper, the local processing is done using the PBSM join algorithm [19].

The final *refinement phase* is required to compare the exact geometry of the spatial objects, and to eliminate duplicates that may be produced as as result of declustering. Recall from Section 3 that the declustering strategies allows a single spatial object to be represented at multiple nodes, either by replicating the object or by creating fragment boxes. Consequently, it is possible for the same pairs of overlapping spatial objects to be independently joined at different nodes. Before producing the final result, these duplicates must be eliminated.

4.1 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 for parallel spatial join algorithms is as shown in Figure 3. Algorithm "A" corresponds to the case when both relations are redeclustered using D-PSS. The operator tree for this algorithm is shown in Figure 4. The first two operators (labeled as operators 1 and 2 in Figure 4) 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**. This sorting step also eliminates duplicate entries. 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 Fig-

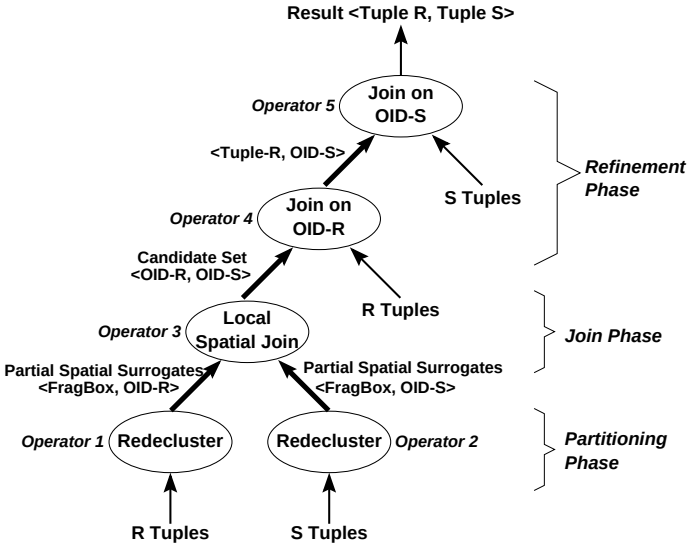

Figure 4: Algorithm A–Shadow Join

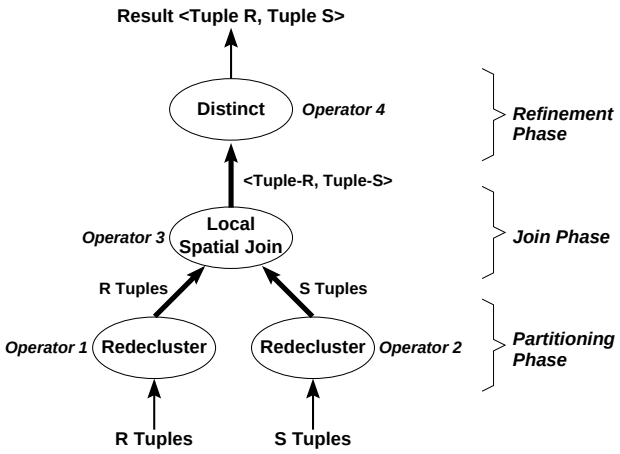

Figure 5: Algorithm B–Clone Join

ure 3). The operator tree for this algorithm is shown in Figure 5. The first two operators (labeled as operator 1 and 2 in Figure 5) redecluster the relations using the D-W declustering policy. Next, a local spatial join operator joins the redeclustered relations. Finally, a distinct operator is used to eliminate duplicates that may be produced as a result of the replication in the declustering steps.

Referring back to Figure 3, 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.

The algorithms in Figure 3 can be adapted if one or both the relations are already declustered on the join attribute. For example, if prior to the join the relations are declustered using D-W, then Algorithm "A" can be run without the redecluster operators. Similarly, if prior to the join only one of the relation is declustered on D-W, then we can either redecluster the other relation using D-W and run Algorithm "A" (without the redecluster operators), or redecluster the other relation using D-PSS and run Algorithm "C" (without the redecluster operator).

For the remainder of this paper, we refer to Algorithm "A" as the *Shadow Join* algorithm, 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* algorithm.

The Shadow join algorithm just described is similar to the parallel spatial join proposed by Zhou, Abel and Truffet [26]. The declustering strategy employed in this 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 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 by that node. In [26], 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 algorithm that is used in the local spatial join.

5. ANALYTICAL MODEL

In this section, we compare the performance of Clone Join and Shadow Join analytically. In the interest of space, we do not present the detailed cost equations here, but redirect the interested reader to the extended version of this paper available at [20]. In our analysis, we examine the effects of the following three parameters:

1. Join Selectivity
2. Replication Probability
3. Precision

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 be replicated when it is declustered. 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 5), the candidate set is produced during the local spatial join, which actually executes in two steps: the filter step and the refinement step. The candidate set is produced right after the filter step of the local join. In Shadow Join (Figure 4), 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 **precision**. It represents how accurate the candidate set is in representing the final result set. A low precision 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 these parameters one at a time. The default values for these parameters are: replication probability = 0.07, precision = 0.2, and join selectivity = $0.5e-6$. These values were chosen

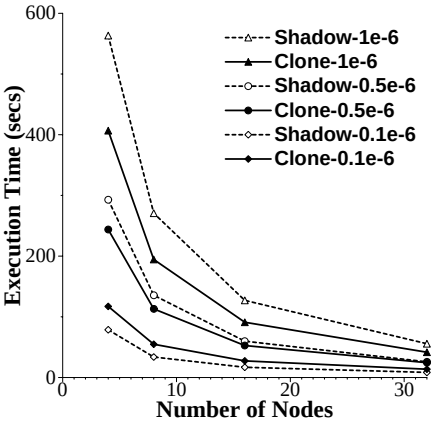

Figure 6: Analytical Model: Effect of Join Selectivity

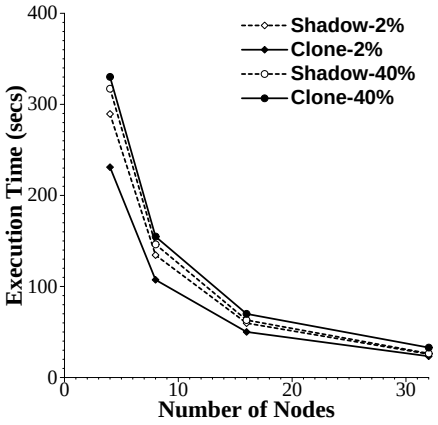

Figure 7: Analytical Model: Effect of Replication Probability

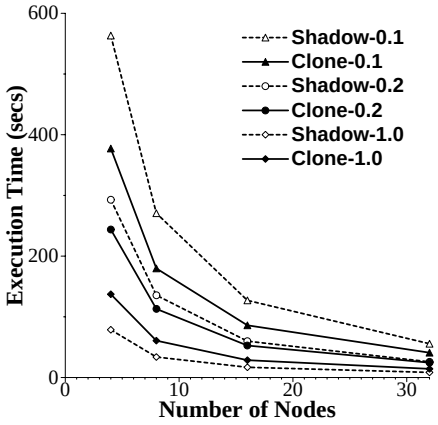

Figure 8: Analytical Model: Effect of Precision

based on actual experiments with the DCW [4] data sets. The cardinality of each input relation is set to 1 million.

5.1 Effect of Join Selectivity

Figure 6 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 4). 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.

5.2 Effect of Replication Probability

Figure 7 plots the effect of the replication probability on the two algorithms for two replication probabilities: 2% and 40%. Increasing the replication probability has little effect on the performance of Shadow Join since this algorithm uses partial spatial surrogates which has a modest replication overhead. Clone Join, on the other hand replicates entire tuples in the partitioning phase and is more susceptible to replication overhead. As the replication probability increases, the replication overhead also increases, and consequently, the performance of Clone Join rapidly degrades.

5.3 Effect of Precision

Figure 8 shows the effect of the precision on the performance of the two join algorithms. For low precision, Clone Join outperforms Shadow Join (compare Clone–0.1 with Shadow–0.1). As the precision increases (compare Clone–1 with Shadow–1), the relative performance of the two algorithms is reversed. When the precision 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 5), and only transfers the pruned set once to the distinct operator. Consequently, lower precision favor the Clone Join algorithm.

5.4 Summary of the Analytical Comparisons

To summarize, the parameters, replication probability, join selectivity and precision, have different effects on the performance of the two algorithms. One difference between the two algorithms is that the Shadow join algorithm uses partial spatial surrogates for replication. Partial spatial surrogates have a very low replication overhead, and, consequently, the Shadow join algorithm 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 5) 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 precision and the join selectivity. A lower precision, or a larger join selectivity implies larger intermediate result sets. Consequently, a low precision or a high join selectivity has a negative effect on Shadow Join's performance, allowing Clone Join to outperform it under these conditions.

6. EXPERIMENTAL EVALUATION

In this section, we compare the performance of Shadow Join and Clone Join based on an actual implementation of these algorithms. These algorithms were implemented inside Paradise, a scalable spatial database system [21].

6.1 Data Sets

The data sets that are used in this benchmark come from the DCW data product [4]. 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 is summarized in Table 1.

	Tuple Count	Size
Drainage	1.73 M	300 MB
Road	0.7 M	100 MB
Rail	0.14 M	18 MB

Table 1: DCW Data Set.

6.2 Testbed 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 node) in the system. All the experiments used 10,000 tiles in the spatial declustering function.

6.3 Experimental Results

6.3.1 Experiment 1

Figure 9 shows the *speedup* of the two algorithms while joining the Drainage relation and the Road relation. 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, Precision = 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.

6.3.2 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 that differed from the previous data set (used in Experiment 1) in terms of the join selectivity characteristics. To obtain the desired effect, 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

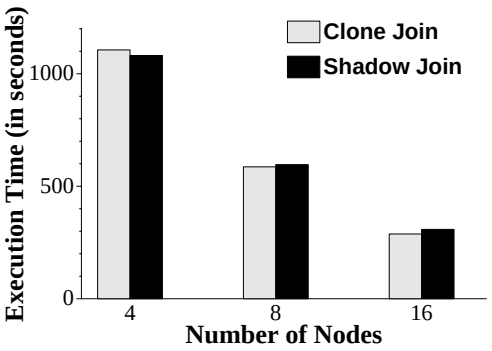

Figure 9: Join Drainage and Roads (Repl. Prob. = 0.07, Precision = 0.20, Join Sel. = $0.53e-6$)

the effect of approximately halving the cardinality of the final result set. For the Shadow Join (refer to Figure 4) every second tuple that is produced by the local join operator (operator 3) is discarded instead of sending it to the next operator. For the Clone Join (refer to Figure 5), 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 10. As predicted by the analytical model, the lower join selectivity favors Shadow Join (see Section 5.1 for an explanation).

6.3.3 Experiment 3: Effect of Precision

We now explore the effects of the precision 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 precision, we duplicate each tuple of the intermediate relation. For the Shadow Join we produce two tuples for every tuple that is produced by the local join operator (operator 3). For the Clone Join 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 cardinality of the candidate lists while keeping the cardinality of the final result set constant. The result of running this experiment is shown in Figure 11. Again, as predicted by the analytical model, a lower precision favors the Clone Join algorithm (see Section 5.3 for an explanation).

6.3.4 Experiment 4: Effect of Replication

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 number of smaller road segments. Each segment is then stored in the database as a separate tuple. This fragmentization is performed 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 is summarized in Table 2.

From Table 2, we observe that the stitching process dra-

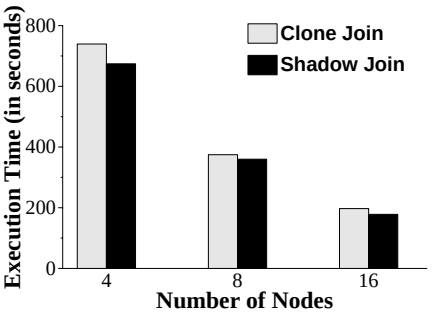

Figure 10: Effect of Join Selectivity: Join Drainage and Roads (Repl. Prob. = 0.07, Precision = 0.20, Join Sel. = $0.27e-6$)

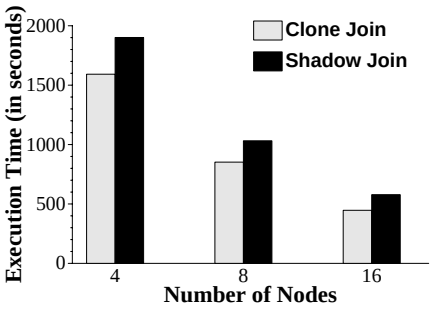

Figure 11: Effect of Precision: Join Drainage and Roads (Repl. Prob. = 0.07, Precision = 0.11, Join Sel. = $0.53e-6$)

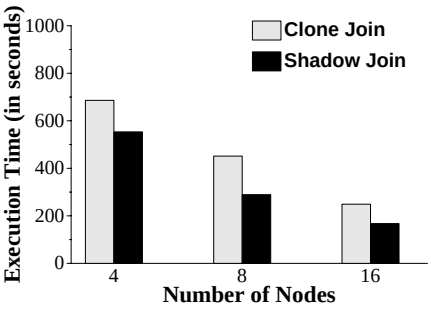

Figure 12: Effect of Replication: Join Stitched-Road and Stitched-Rails (Repl. Prob. = 0.18, Precision = 0.23, Join Sel. = $1.3e-6$)

	# Tuples	Size	Avg. stitched length	Avg. original length
Drainage	0.99 M	243 MB	20.18 km	11.61 km
Road	0.23 M	67 MB	45.28 km	14.99 km
Rail	0.04 M	12 MB	41.05 km	12.28 km

Table 2: Stitched DCW Data Set.

matically 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 12 shows the results of this experiment. The results shows that when the replication probability is high, Shadow Join outperforms Clone Join (again, for the same reasons described in Section 5.2).

7. CODING COMPLEXITY

The experimental and the analytical experiments show that neither the Shadow or the Clone join algorithm is superior in all cases. The relative performance of the two algorithms depends on the data characteristics like the replication probability, the precision, and the join selectivity. Fortunately, both the Shadow and the Clone join algorithms share a number of software modules, and also reuse modules that already exist in conventional database systems. This makes it easier to implement both these algorithms. The query optimizer can then use the analytical cost formulae to pick the cheapest algorithm for each spatial join operation.

First, consider the coding complexities of the "Redecluster" operator in the Shadow Join (operator 1 and 2 in Figure 4). A parallel database system must have some kind of a redecluster operator, with a list of declustering policies. In a relational system the common declustering policies are hash and round-robin. The Clone Join requires adding D-PSS as a declustering strategy. The code for this requires computing the MBR (minimum bounding rectangle) of a spatial object, and then computing the intersection of the MBR with tiles in the spatial declustering policy. With full error checking and the ability to reuse this functionality in other modules (like the internal partitioning for the local spatial join processing), this part of the code is a few hundred lines of C++ code in Paradise. The "Redecluster" operator in Clone Join (operator 1 and 2 in Figure 5) is even simpler to implement as it simply requires replicating tuples. This module is less than a hundred lines of C++ code in Paradise. The "local spatial join" is required both by the Clone Join and the Shadow Join (operator 3 in both Figure 4 and Figure 5). This part of the code is slightly over a thousand lines in Paradise. The distinct operator in the Clone Join (operator 4 in figure 5) is the same as a distinct operator that is available in any conventional database system. The "Join on OID-R" operator (operators 5 and 6 in figure 4), requires sorting tuples based on an OID attribute, and then sequentially fetching the tuples and evaluating the join predicate. The sorting part of this operator reuses the conventional sort utility that is available in Paradise. The additional code needed for this operator (beyond the sorting) is a few hundred lines in Paradise.

8. CONCLUSIONS

In this paper, we first proposed two strategies for declustering spatial data in a parallel database system. These techniques are based on partitioning the underlying space into regions, and mapping these regions to nodes. Spatial declustering strategies require some form of redundancy, and both these strategies employ some form of replication. One strategy replicates entire tuples, whereas the other strategy replicates an approximation of the spatial attribute. Based on these alternative declustering policies we then explored the design space for parallel spatial join algorithms, and identified two key algorithms in the design space - the Clone Join and the Shadow Join. Finally, we presented results obtained from analytical modeling and an actual implementation of these algorithms. Various experiments, using real geographic data, were run on a of cluster of PCs. The experimental results show that the parallel spatial join algorithms exhibit good speedup characteristics. The experimental results also demonstrate that the relative performance of the two algorithms is dependent on the characteristics of the data set, and the selectivity of the join. Fortunately, in an actual implementation, it is possible for both these algorithms to share a number of software modules, and reuse modules that already exist in a conventional database system. This makes it easy to implement both algorithms. The query optimizer can then pick the best algorithm for a given spatial join operator. In aiding the query optimizer to pick the best algorithm, it is essential to capture the characteristics of the data. Spatial sampling techniques, like those

described in [17] can potentially be used for this purpose.

9. REFERENCES

[1] D. Abel, B. C. Ooi, K. Tan, R. Power, and J. X. Yu. "Spatial Join Strategies in Distributed Spatial DBMS". In *Proceedings of 4th Intl. Symp. On Large Spatial Databases*, pages 348—367, 1995.

[2] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. "Efficient Processing of Spatial Joins Using R-trees". In *Proceedings of the 1993 ACM-SIGMOD Conference*, Washington, DC, May 1993.

[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using R-trees. In *Proceedings of the 12th International Conference on Data Engineering*, pages 258–265, Washington - Brussels - Tokyo, Feb. 1996. IEEE Computer Society.

[4] *"VPFView 1.0 Users Manual for the Digital Chart of the World"*. Defense Mapping Agency, July 1992.

[5] D. J. DeWitt and J. Gray. "Parallel Database Systems: The Future of Database Processing or a Passing Fad?". *Communication of the ACM*, June, 1992.

[6] O. Günther. "Efficient Computation of Spatial Joins". In *IEEE TKDE*, 1993.

[7] R. H. Güting and W. Shilling. "A Practical Divide–and–Conquer Algorithm for the Rectangle Intersection Problem". In *Information Sciences*, volume 42, 1987.

[8] A. Gutman. "R-trees: A Dynamic Index Structure for Spatial Searching". In *Proceedings of the 1984 ACM-SIGMOD Conference*, Boston, Mass, June 1984.

[9] E. G. Hoel and H. Samet. "Performance of Data-Parallel Spatial Operations". In *Proceedings of the 20th VLDB Conf.*, pages 156–167, Santiago, Chile, Sept. 1994.

[10] E. G. Hoel and H. Samet. "Benchmarking Spatial Join Operations with Spatial Output". In *Proceedings of the 21st VLDB Conf.*, Zurich, Switzerland, Sept. 1995.

[11] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. "Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations". In *Proceedings of the 23st VLDB Conf.*, pages 396–405, Athens, Greece, Aug. 1997.

[12] I. Kamel and C. Faloutsos. "Parallel R–Trees". In *Proceedings of the 1992 ACM-SIGMOD Conference*, San Diego, California, June 1992.

[13] M. Kitsuregawa and Y. Ogawa. "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC) ". In *Proceedings of the 16th VLDB Conf.*, pages 210—221, 1990.

[14] N. Koudas, C. Faloutsos, and I. Kamel. "Declustering Spatial Databases on a Multi—Computer Architecture". In *EDBT*, pages 592–614, 1996.

[15] M. L. Lo and C. V. Ravishankar. "Spatial Joins Using Seeded Trees". In *Proceedings of the 1994 ACM-SIGMOD Conference*, Minneapolis, May 1994.

[16] M. L. Lo and C. V. Ravishankar. "Spatial Hash–Joins". In *Proceedings of the 1996 ACM-SIGMOD Conference*, Montreal, Canada, June 1996.

[17] F. Olken and D. Rotem. "Sampling from Spatial Databases". In *Proc. of the 9th International Conference on Data Engineering*, pages 199–208, Apr. 1993.

[18] J. A. Orenstein. "Spatial Query Processing in an Object–Oriented Database System". In *Proceedings of the 1986 ACM-SIGMOD Conference*, 1986.

[19] J. M. Patel and D. J. DeWitt. Partition based spatial–merge join. In *Proceedings of ACM SIGMOD 1996*, pages 259–270, June 1996.

[20] J. M. Patel and D. J. DeWitt. "Clone Join and Shadow Join: Two Parallel Algorithms for Executing Spatial Join Operations". Technical Report, University of Wisconsin, CS-TR-99-1403, Aug. 1999.

[21] J. M. Patel, J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. E. Hall, K. Ramasamy, R. Lueder, C. Ellman, J. Kupsch, S. Guo, D. J. DeWitt, and J. F. Naughton. "Building a Scaleable Geo—Spatial DBMS: Technology, Implementation, and Evaluation.". In *Proceedings of the 1997 ACM-SIGMOD Conference*, pages 336–347, Tucson, Arizona, USA, May 1997.

[22] S. Shekhar, S. Ravada, V. Kumar, D. Chubb, and G. Turner. "Declustering and Load-Balancing Methods for Parallelizing Geographic Information Systems". In *IEEE TKDE*, volume 10(4), pages 632–655, 1998.

[23] SpaceImaging Corp., Lanham, MD. *Space Imaging Catalog of Products and Services*, Feb. 1999.

[24] K.-L. Tan and J. X. Yu. "A Performance Study of Declustering Strategies for Parallel Spatial Databases". In *DEXA*, pages 157–166, London, United Kingdom, Sept. 1995.

[25] C. B. Walton, A. G. Dale, and R. Jenevein. "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins". In *Proceedings of the 17th VLDB Conf.*, pages 537—548, Barcelona, Spain, Sept. 1991.

[26] X. Zhou, D. J. Abel, and D. Truffet. "Data Partitioning For Parallel Spatial Join Processing". In *Proceedings of 5th Intl. Symp. On Large Spatial Databases*, pages 178—196, Berlin, Germany, July 1997.