Clone Join and Shadow Join: Two Parallel Algorithms for Executing Spatial Join Operations

Jignesh Patel
David DeWitt

Technical Report #1399

June 1999

Clone Join and Shadow Join: Two Parallel Algorithms for Executing Spatial Join Operations*

Jignesh M. Patel David J. DeWitt

Computer Sciences Department
University of Wisconsin, Madison
{jignesh, dewitt}@cs.wisc.edu

Abstract

With the growing popularity of spatial applications, there has been a significant increase in the use of database systems for storing and querying spatial data. Spatial data is now readily available from a variety of sources including government mapping agencies, commercial sources, satellite images, and simulation outputs. As this trend continues, applications continue to execute increasingly complex queries on large and larger volumes of spatial data. As can be expected, these complex spatial queries frequently involve joining two data sets based on some spatial relationship between objects in the two data sets. This operation is called a spatial join, and like its relational counterpart, is an expensive operation. Consequently, spatial database systems must employ efficient spatial join algorithms. In the past, many algorithms have been proposed for evaluating a 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 storing and querying large volumes of spatial data. We first propose and analyze some strategies for storing spatial data in a parallel database system. In this paper, we focus primarily on the static space partitioning approach that first statically partitions the underlying space, and then maps these partitions to processors. We propose a number of spatial join algorithms based on these declustering strategies. Two algorithms are identified as the primary algorithms in this design space. We develop analytical cost models for these two algorithms, and, using the analytical model we identify key parameters that influence the performance of these join algorithms. Finally, using real data sets and an actual implementation, we test the performance of these algorithms. The experiments show that both algorithms can effectively exploit parallelism.

1 Introduction and Motivation

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 on a centralized system have been proposed [Ore86, BKS93, Gün93, HJR97, GS87, LR94, HS95, LR96, PD96]. 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. Obviously, such data sets for the entire world, and at higher resolutions will be much larger. In the near future, we will probably see even larger data sets. High resolution satellite images are now becoming available for commercial purposes at a very economical price [Spa99]. A variety of applications that make use of these easily available high resolution satellite images are now starting to emerge [Ter99]. 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

*The 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.

easily generate very large spatial data sets. With this inevitable increase in the size of spatial data sets, 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 try using parallelism to store and query large spatial data sets. 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, like polygons and polylines, actually span 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 either 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 partitioning spatial data, and consequently, in a parallel system, requires novel algorithms for operations like the spatial join.

This paper explores various algorithms for evaluating the spatial join operation in a parallel spatial database system. First, strategies for partitioning spatial data are examined. Spatial data can be partitioned 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 operates by inserting the spatial objects into a spatial index, like an R-tree [Gut84], and mapping the leaf nodes of the R-tree to nodes in the parallel system. In [ZAT97] 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. This paper proposes and analyzes two different declustering strategies. 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. Analytical cost models, that can be used in a query optimizer, are developed for these algorithms. The analytical model identifies the parameters that influences the performance of the two algorithms. The two spatial join algorithms are also implemented in Paradise, a parallel spatial database system, and this paper also present the results of this implementation. The experiments show that both these algorithms exploit parallelism effectively.

2 Related Work

While parallel spatial databases are a relatively new area, in the last few years some attention has been directed towards various aspects of implementing a parallel spatial database system. In this section we review the related work in this area.

In [TY95], a few techniques for declustering spatial data is proposed. This is followed by a performance study evaluating 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. In [KF92], the authors 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 [Gut84] 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 in [KFK96] to decluster spatial data on a shared-nothing architecture. One node in the system is dedicated for performing the bookkeeping associated with mapping the leaves of the R-tree to the other nodes in the system. Again, the scope of the study is limited to queries with spatial selections. In [HS94], the authors examine the use of PMR Quadtrees, R+-trees, and R-trees for evaluating the spatial join on a special-purpose platform (Thinking Machines, CM-5).

No I/O is considered in the paper, as all the data is always resident in main memory. Similarly, in [BKS96] two R*-trees are used for performing a spatial join in a shared-disk environment.

Various spatial join algorithms have been proposed for evaluating the spatial join in a centralized spatial database system. Drawing inspiration from the relation hash-based join algorithms, many of these algorithm divide large spatial relations (relations that are bigger in size than the available memory) into a number of smaller partitions. A spatial partitioning function divides both the spatial relations into a number of smaller partitions, such that each partition is small enough to fit in memory. The join is then evaluated by pair-wise joining of the smaller partitions. The spatial partitioning function that have been proposed for the centralized spatial join algorithms fall into two categories: dynamic spatial partitioning function and static spatial partitioning function. A dynamic partitioning function inserts the spatial objects into a spatial index, and maps the leaf nodes of the spatial index to partitions. The centralized spatial join algorithms proposed in [LR94, HS95, LR96, KS97, BKS93, Gün93, HJR97] use a dynamic spatial partitioning function. A static partitioning function divides the underlying space into regions, and maps the regions to partitions. The centralized spatial join algorithm proposed in [PD96] uses a static spatial partitioning function. The potential of using the static spatial partitioning function for declustering spatial data in a parallel environment, and for parallel spatial join evaluation is also briefly discussed in [PD96].

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. Using the spatial partitioning function we proposed in [PD96], the data is distributed to partitions by first dividing the space into cells. The cells are then mapped to partitions. 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.

Our paper differs from the study in [ZAT97] as we propose two algorithms for parallel spatial joins, and evaluate them on a real system using a number of large data sets.

3 Spatial Declustering Techniques

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 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 such declustering techniques.

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 1 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. 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. Let us first examine this declustering strategy where tuples with spatial attributes spanning multiple regions are replicated at all the nodes corresponding to the overlapping region.

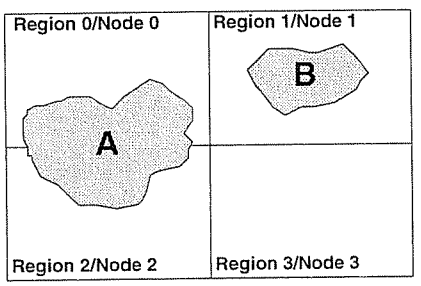

Figure 1: Declustering of Polygon Data

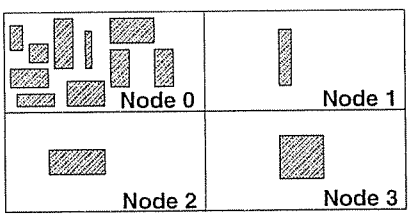

Figure 2: Data Distribution Skew

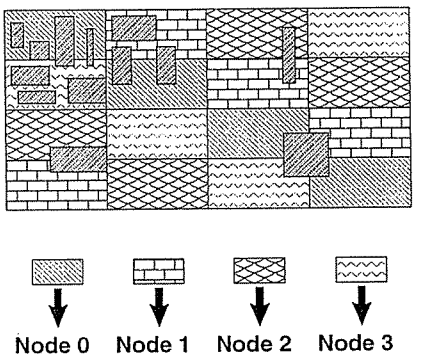

Figure 3: Tiling to Reduce Data Distribution Skew

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 (as illustrated by Figure 2). 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]. 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 3 shows the same relation as in Figure 2 being declustered across 4 nodes using 16 tiles, with the tiles being mapped to nodes using a hash function. This declustering strategy is similar to the spatial partitioning function that is used internally in the PBSM spatial join algorithm [PD96]. There it was shown that hashing is superior to round robin. Consequently, in this paper we only consider hashing.

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 2, 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 tuples will have spatial attributes that overlap tiles that are mapped to different nodes, thereby increasing the replication cost. To quantify these tradeoff, 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. The result of declustering this data set is shown in Figure 4. 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 4).

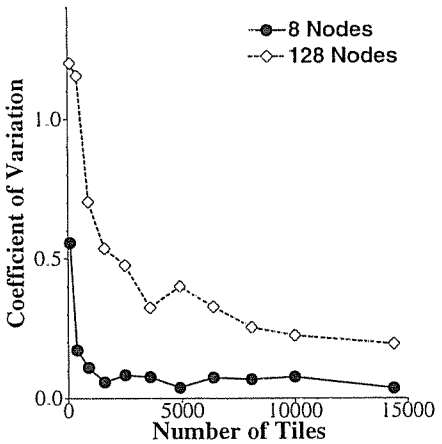

Figure 4: Spatial Distribution Skew: DCW Data

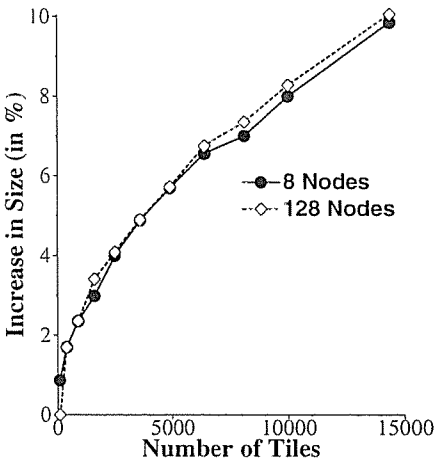

Figure 5: Replication Overhead: DCW Data

Figure 5 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 small for this data set. However, the replication overhead could be much higher for a data set with different data 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 6 which shows the data distribution skew for this data set. Similarly to the DCW landuse data set (see Figure 4), 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 7 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 is very high. For example, in the 5000 tile, 128 node configuration, the size of the declustered relation on disk increases by 300%. Figure 8 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.

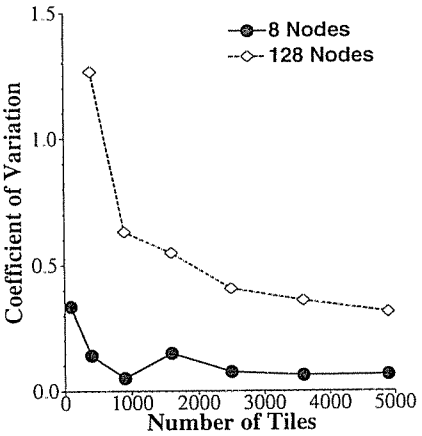

Figure 6: Spatial Distribution Skew: Sequoia Landuse Data Set

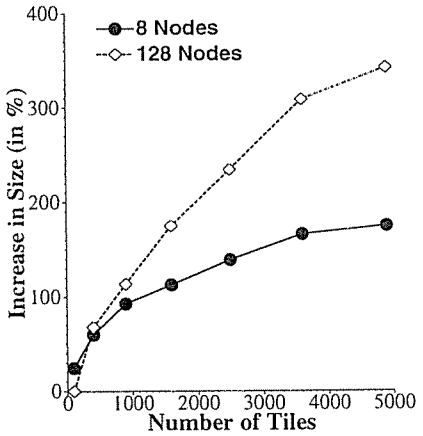

Figure 7: Replication Space Overhead with the Sequoia Landuse Data Set

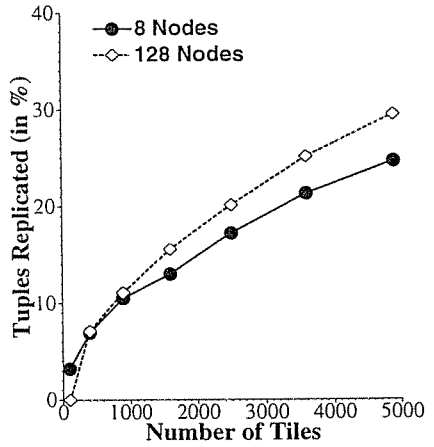

Figure 8: Replication of Tuples with the Sequoia Landuse Data Set

3.2 Partial Spatial Surrogates

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 OID of the tuple and the minimum bounding rectangle (MBR) of that 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 9 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 6, 7, and 8. Figure 10 shows the space overhead for this data set using partial spatial surrogates. Comparing Figure 10 with Figure 7 shows that the use of partial spatial surrogates dramatically reduces the space overhead associated with spatial declustering.

To summarize, we have two spatial declustering strategies. In the first strategy, an entire tuple is replicated when the declustering spatial attribute overlaps tiles that are mapped to multiple nodes. For the remainder of this paper, this declustering strategy is referred to as **D-W** ("decluster using whole tuple replication"). The other strategy is to create partial spatial surrogates when the declustering spatial attribute overlaps tiles that are mapped to multiple nodes. This strategy is referred to as **D-PSS** ("decluster creating spatial surrogates").

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

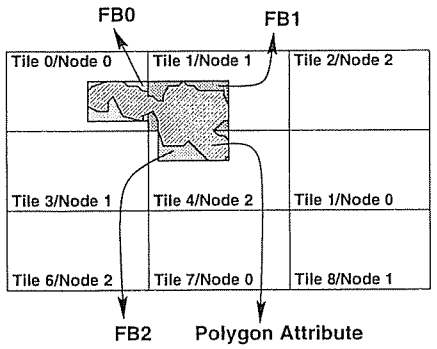

Figure 9: Declustering Using Spatial Surrogates

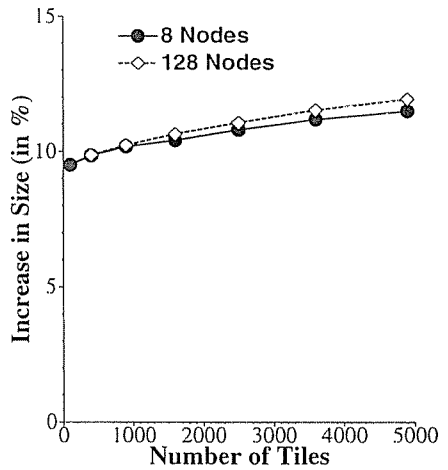

Figure 10: Declustering Overhead of the Sequoia Landuse Data Set Using Partial Spatial Surrogates

2. Join Phase
3. Refinement Phase

In the *partitioning phase*, the two relations being joined are redeclustered on their (spatial) join attributes. If, before the join, 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 of the centralized join algorithms [Ore86, BKS93, Gün93, HJR97, GS87, LR94, HS95, LR96, PD96]. In this paper, the local processing is done using the PBSM join algorithm [PD96].

The final *refinement phase* is required for two reasons. First, if one of the inputs in the partitioning phase is redeclustered using D-PSS (see Section 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 redeclustered using D-W (see Section 3.1). For example consider Figure 11, 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 final refinement phase, a distinct operator is used to eliminate one of these result pairs.

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 of the parallel spatial join algorithms is as shown in Figure 12. Algorithm "A" corresponds to the case when both relations are redeclustered using D-PSS. The operator tree for this algorithm is shown in Figure 13. The first two operators (labeled as operators 1 and 2 in Figure 13) 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

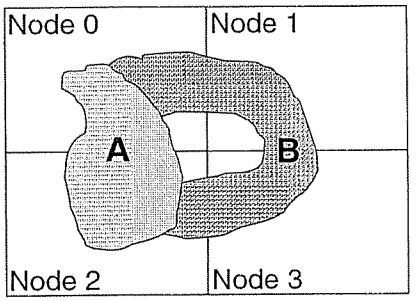

Figure 11: Need for Duplicate Elimination

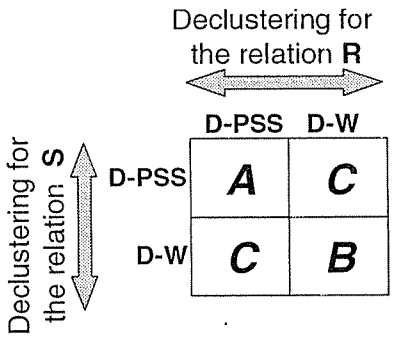

Figure 12: Design Space of Spatial Join Algorithms

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**.

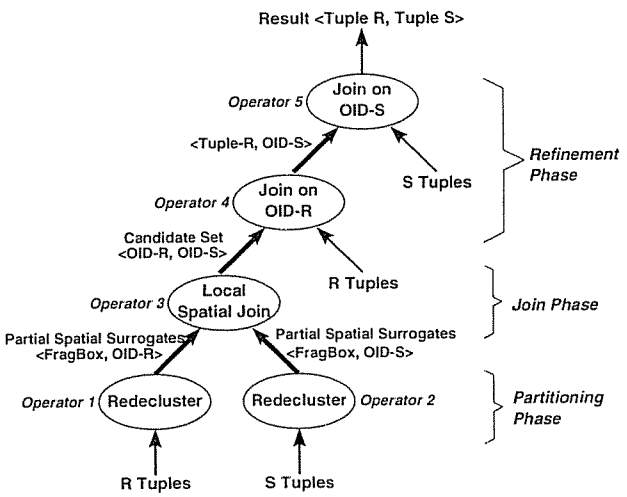

Figure 13: Join Using Partial Spatial Surrogates For Redeclustering (Algorithm A—Shadow Join)

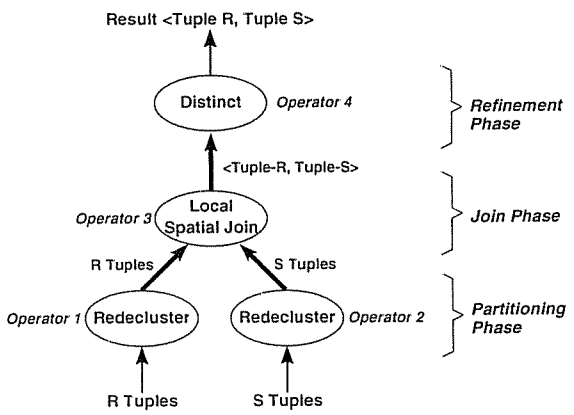

Figure 14: Join Using Whole Tuple Redeclustering (Algorithm B—Clone Join)

Now consider Algorithm "B" in the design space (see Figure 12). The operator tree for this algorithm is shown in Figure 14. The first two operators (labeled as operator 1 and 2 in Figure 14) 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 11), a distinct operator ensures that the replication does not produce any additional duplicates.

Referring back to Figure 12, 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 12 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 redecluster 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 redecluster operators), or redecluster the other relation using D-PSS and run Algorithm "C" (without the redecluster operator).

For the rest of this paper, 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 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 [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 algorithm that is used in the local spatial join. 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. Furthermore, their algorithm always assumes that all the data fits in main memory.

5 Analytical Model

Let us now compare the performance of the spatial join algorithms outlined in Figure 12. 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 these analytical models is to understand the relative performance of the two algorithms, and to produce cost functions that can be plugged into a cost-based optimizer.

5.1 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**. Let $|M|$ denote the size of main memory in pages. The system parameters for the model are shown in Table 1, and the parameters that characterize the data sets are shown in Table 2. 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 2 is the factor by which the cardinality of a declustered relation increases after a relation is declustered. This factor is always greater than 1. The join selectivity in Table 2 is the ratio of the cardinality of the output relation to the product of the cardinalities of the input relations.

5.2 Cost Formula for the Clone Join

Let us first derive the cost formula for the Clone Join (see Figure 14). The cost of redeclustering a relation **R** based on the whole declustering strategy (D-W) is:

$$\begin{aligned} C_{DW}(R) &= |R| \times C_{I/O} && (\textit{Cost of reading the input}) \\ &+ ||R|| \times C_{FragBox} && (\textit{Cost of fragmenting tuples}) \\ &+ |R| \times F_{Repl} \times C_{Net} && (\textit{Send tuples to the next operator}) \end{aligned}$$

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

Symbol	Description	Value
$C_{I/O}$	Cost of a disk I/O	2 ms/page
C_{Net}	Cost of sending a page over the network	0.67 ms/page
$C_{FragBox}$	Cost of producing a partial spatial surrogate	900 cycles
C_{Cmp}	Cost of comparing two integers	1 cycle
C_{Hash}	Cost of hashing on multiple attributes and inserting into a hash table	200 cycles
$C_{OidLookup}$	Cost of looking up the node information from an OID	5 cycles
C_{OidCmp}	Cost of comparing (for sorting) two OIDs	20 cycles
C_{EM}	Cost of examining two spatial attributes for overlap or containment	20000 cycles
S_{Page}	Size of a page	8192 bytes
S_{SpSur}	Size of a partial spatial surrogate	48 bytes
	CPU speed	133 MHz

Table 1: System Parameters.

Symbol	Description	Value
F_{Repl}	Replication Factor	variable
F_{Join}	Join Selectivity	variable
F_{Cand}	Ratio of candidate list cardinality to final result cardinality	variable
X_{Ovlp}	Average number of rectangles that intersect a vertical plane sweep line	100
S_{Tuple}	Average size of a tuple	180 bytes

Table 2: Data Characteristics.

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 are too large to fit in memory, then they are partitioned, and the partitions are joined [PD96]. 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:

$$
\begin{aligned}
C_{RectJoin}(R,S) &= (|R|+|S|)\times C_{I/O} && (Cost\ of\ reading\ the\ inputs)\\
&+ ||R||\times C_{FragBox} + ||S||\times C_{FragBox} && (Cost\ of\ extracting\ MBR)\\
&+ [(|R_S|+|S_S|) && \\
&\quad -min(|R_S|+|S_S|,|M|-2P)]\times 2\times C_{I/O} && (Cost\ of\ partitioning)\\
&+ ||R||\times F_{Repl}\times log(||R_P||)\times C_{Cmp} && (Cost\ of\ sorting\ rectangles)\\
&+ ||S||\times F_{Repl}\times log(||S_P||)\times C_{Cmp} && (Cost\ of\ sorting\ rectangles)\\
&+ (||R||+||S||)\times X_{Ovlp}\times C_{Cmp} && (Cost\ of\ the\ plane-sweep)
\end{aligned}
$$

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: $|R_S| = \frac{||R||\times S_{SpSur}\times F_{Repl}}{S_{Page}}$

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 [PD96]. The cost of this step is:

$$\begin{aligned}
C_{Refinement} &= 2 \times [|C| - min(|C|, |M| - \sqrt{|C|})] \times C_{I/O} && (I/O\ cost\ for\ sorting) \\
&+ ||C|| \times log(||C||) \times C_{OidCmp} && (CPU\ cost\ for\ sorting) \\
&+ |R| \times C_{I/O} && (Read\ tuples\ of\ R) \\
&+ |S| \times \frac{||C|| \times (S_{Tuple} + S_{SpSur})}{S_{Page} \times |M|} \times C_{I/O} && (Multiple\ passes\ for\ fetching\ S\ tuple) \\
&+ ||C|| \times C_{EM} && (Evaluate\ the\ join\ predicate) \\
&+ F_{Join} \times (||R|| \times ||S||) \\
&\quad \times F_{Ovlp} \times \frac{2 \times S_{Tuple}}{S_{Page}} \times C_{Net} && (Send\ tuples\ to\ the\ next\ operator)
\end{aligned}$$

Here C is the intermediate result consisting of the list of OID pairs. The number of tuples in this relation is $F_{Join} \times (||R|| \times ||S||) \times F_{Ovlp} \times F_{CandList}$, and the size of each tuple is the same as the 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 14). The distinct operator uses 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: $|T| = \frac{(||R|| \times ||S||) \times F_{Ovlp} \times F_{Join} \times 2 \times S_{Tuple}}{S_{Page}}$

The cost of the distinct operator is:

$$\begin{aligned}
C_{Distinct} &= 2 \times [|T| - min(|T|, |M| - \sqrt{|T|})] \times C_{I/O} && (Partitioning\ cost) \\
&+ (||R|| \times ||S||) \times F_{Ovlp} \times F_{Join} \times C_{Hash} && (Cost\ of\ hashing)
\end{aligned}$$

Thus the total cost of Clone Join is:

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

5.3 Cost Formula for the Shadow Join

We now derive the cost formula for the Shadow Join (Figure 13).The cost of redeclustering a relation **R** creating partial spatial surrogates (D-PSS) is:

$$\begin{aligned}
C_{DPSS}(R) &= |R| \times C_{I/O} && (Cost\ of\ reading\ the\ input) \\
&+ ||R|| \times C_{FragBox} && (Cost\ of\ fragmenting\ tuples) \\
&+ \frac{||R|| \times S_{SpSur}}{S_{Page}} \times F_{Repl} \times C_{Net} && (Send\ tuples\ to\ next\ operator)
\end{aligned}$$

The cost of joining the two sets of spatial surrogates is the same $C_{RectJoin}$ 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 $(||R|| \times ||S||) \times F_{Join} \times F_{Repl} \times F_{Cand}$, and the cost of redeclustering this set is:

$$\begin{aligned}
C_{DOID}(R) &= (||R|| \times ||S||) \times F_{Join} \times F_{Repl} \times F_{Cand} \times F_{OidLookup} && (Look\ up\ the\ node) \\
&+ \frac{(||R|| \times ||S||) \times F_{Join} \times F_{Repl} \times F_{Cand} \times 2 \times S_{SpSur}}{S_{Page}} \times C_{Net} && (Send\ tuples\ to\ the\ next\ operator)
\end{aligned}$$

The last two operators in Figure 13 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:

$$\begin{aligned}
C_{OidSort}(T) &= 2 \times [|T| - min(|T|, |M| - \sqrt{|T|})] \times C_{I/O} && (I/O\ cost\ for\ sorting) \\
&+ ||T|| \times log(||T||) \times C_{OidCmp} && (CPU\ cost\ for\ sorting)
\end{aligned}$$

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 13) this factor is equal to F_{Repl}, and in the second OID join operator (operator 5 in Figure 13) this factor is equal to 1. The number of tuples in the relation **T** is $(||R|| \times ||S||) \times F_{Join} \times F_{Repl} \times F_{Cand}$ in the first join operator, and $(||R|| \times ||S||) \times F_{Join} \times F_{Cand}$ in the second join operator. The average size of a **T** tuple is $2 \times S_{SpSur}$ in the first OID join operator, and $S_{SpSur} + S_{Tuple}$ 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:

$$\begin{aligned} C_{Shadow}(R,S) &= C_{DPSS}(R) + C_{DPSS}(S) + C_{RectJoin}(R,S) + C_{DOID}(R) \\ &+ C_{OidSort}(T^1) + C_{Fetch}(T^1) + C_{OidSort}(T^2) + C_{Fetch}(T^2) + C_{RefinementPSS} \end{aligned}$$

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 2, 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 14), the candidate set is produced during the local spatial join (as explained in Section 5.2, this step is executed right after the filter step of the local join). In Shadow Join (Figure 13), 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 = $0.5e-6$. 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.

5.5 Effect of Join Selectivity

Figure 15 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 13). 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.

5.6 Effect of Replication Probability

Figure 16 shows the effect of the replication probability on the two algorithms. The predicted performance of Shadow Join and Clone Join is plotted for replication probabilities of 2% and 40%. Increasing the replication

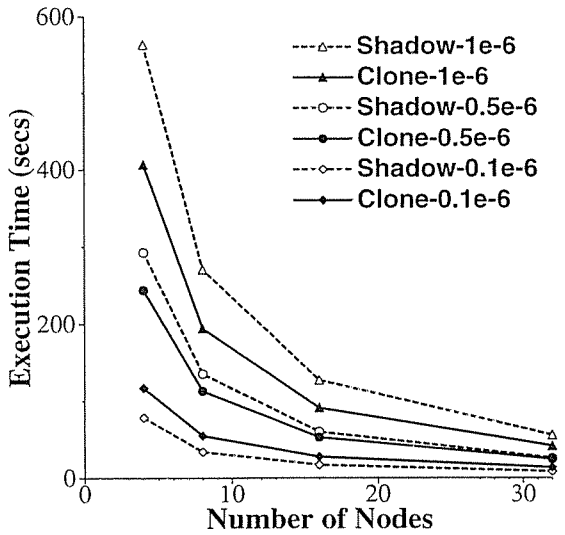

Figure 15: Analytical Model: Effect of Join Selectivity

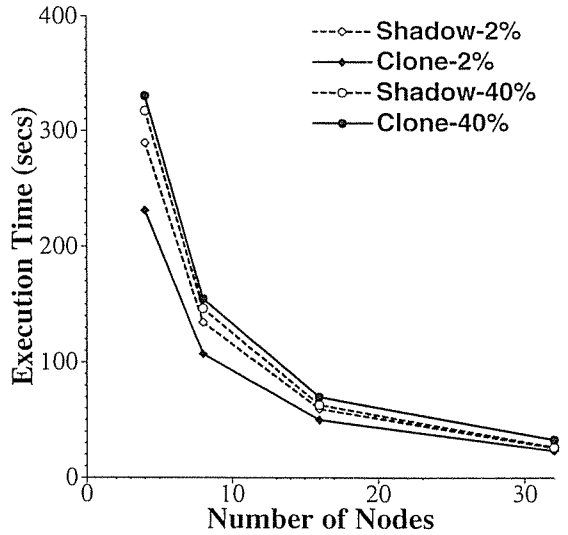

Figure 16: Analytical Model: Effect of Replication Probability

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 10). 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.

5.7 Effect of Fertility Ratio

Figure 17 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 (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 14), and only transfers the pruned set once to the distinct operator. Consequently, lower fertility ratios favor the Clone Join algorithm.

5.8 Summary of the Analytical Performance Comparisons

To summarize, the parameters, replication probablilty, join selectivity and fertility ratio, 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 14) 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

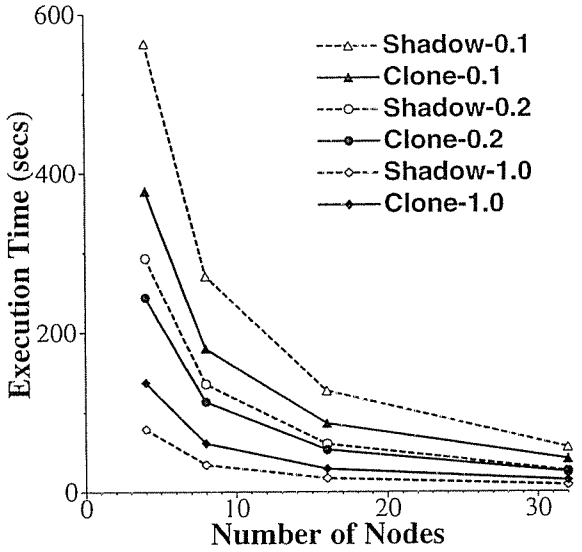

Figure 17: Analytical Model: Effect of Fertility Ratio

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.

6 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 spatial database system [PYK+97].

6.1 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 3.

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

Table 3: DCW Data Set.

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

6.3 Validation of the Analytical Model

We tried validating the analytical model that was presented Section 5 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 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.

6.4 Experimental Results

6.4.1 Experiment 1

Figure 18 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, 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.

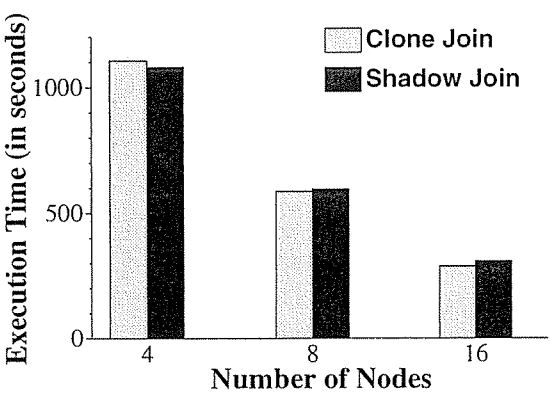

Figure 18: Join Drainage and Roads (Replication Probability = 0.07, Fertility Rate = 0.20, Join Selectivity = $0.53e-6$)

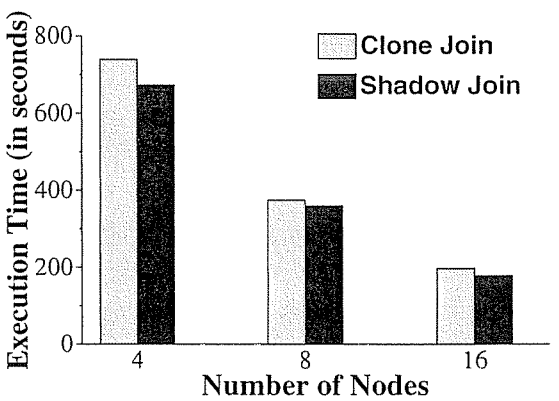

Figure 19: Effect of Join Selectivity: Join Drainage and Roads (Replication Probability = 0.07, Fertility Rate = 0.20, **Join Selectivity** = $0.27e-6$)

6.4.2 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 the effect of approximately halving the cardinality of the final result set. For the Shadow Join (refer to Figure 13) 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 14), 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 19. As predicted by the analytical model, the lower join selectivity favors Shadow Join (see Section 5.5 for an explanation).

6.4.3 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 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 20. Again, as predicted by the analytical model, a lower fertility ratio favors the Clone Join algorithm (see Section 5.7 for an explanation).

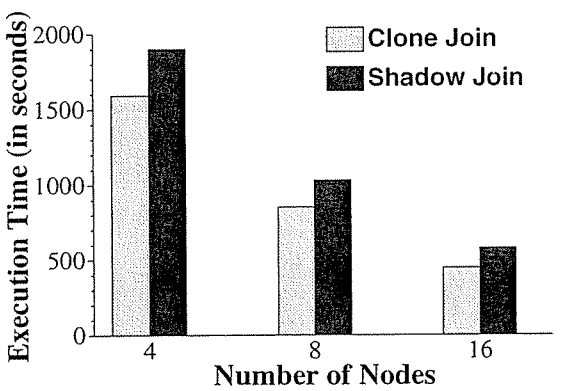

Figure 20: Effect of Fertility Ratio: Join Drainage and Roads (Replication Probability = 0.07, **Fertility Rate = 0.11**, Join Selectivity = $0.53e-6$)

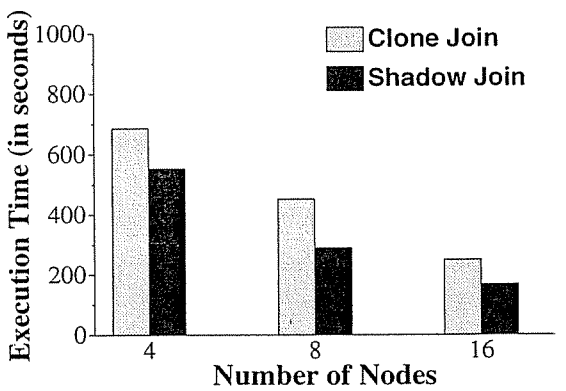

Figure 21: Effect of Replication: Join Stitched-Road and Stitched-Rails (**Replication Probability = 0.18**, Fertility Rate = 0.23, Join Selectivity = $1.3e-6$)

6.4.4 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 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 are summarized in Table 4.

From Table 4, 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 21 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 5.6).

	Tuple Count	Size	Average length of a stitched feature	Average feature length in the original data set
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 4: Stitched DCW Data Set.

6.4.5 Experiment 5: Multiple Finer Approximations

Both the Shadow Join and the 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 (MBR) 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 set that can then be pruned by examining the actual spatial attributes. However, an approximation like the MBR can be a poor representation of the spatial attribute. As an example, consider the river shown in Figure 22. 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 22. The road overlaps with 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 to the candidate set. Another way of approximating the same river attribute is shown in Figure 23. Here, instead of approximating the river by one rectangle (the MBR), it is approximated by five smaller rectangles. These rectangles are called fragment boxes, and are a better approximation of the river tuple. 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.

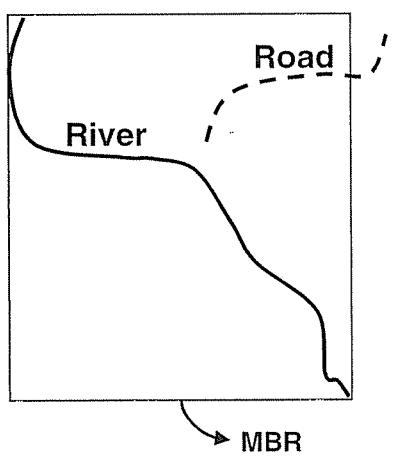

Figure 22: One Approximation per Tuple

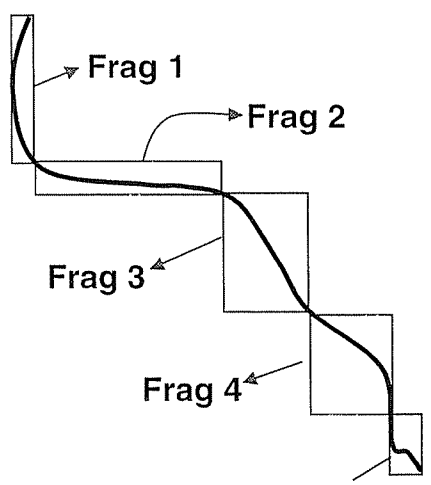

Figure 23: Multiple Approximations

A similar problem occurs in centralized spatial join algorithms, since many centralized spatial join algorithms first join using the MBR 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

Algorithm	4 nodes	8 nodes	16 nodes
Clone	1105.9	586.2	287.8
Clone-MA	1104.0	583.7	295.0
Shadow	1080.9	596.0	308.3
Shadow-MA	1072.7	585.3	306.0

Table 5: Effect of Multiple Finer Approximations, Join Drainage and Road

Algorithm	4 nodes	8 nodes	16 nodes
Clone	686.1	451.1	249.1
Clone-MA	576.0	337.0	201.8
Shadow	553.3	289.2	167.4
Shadow-MA	425.2	213.2	128.9

Table 6: Effect of Multiple Finer Approximations, Join Stitched Road and Rail

approximations on the Shadow Join and the Clone Join algorithms. 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 twice that of the threshold limit is approximated by 20 small rectangles.

Table 5 shows the results of joining the Drainage relation and the Road relation (the same data set as in Experiment 1). In this table the algorithm labeled "Clone-MA" and "Shadow-MA" are the Clone and the Shadow join algorithms using multiple approximations. From this table, we observe that using the multiple approximations has very little effect on the performance of the join algorithms. The benefit of a better approximation is offset by the cost of actually forming these approximations

Table 6 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 6.4.3).

7 Coding Complexity of the Two Algorithms

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 fertility ratio, 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 13). 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 14) 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 13 and Figure 14). This part of the code is slightly over a thousand lines in Paradise. The distinct

operator in the Clone Join (operator 4 in figure 14) 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 13), 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 avaliable 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 have addressed some of the issues that a parallel spatial database system faces. First, we proposed and examined 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. We developed analytical cost models for these two algorithms, and used these models to identify the parameters that influence the performance of the two algorithms. Finally, we presented results obtained from actual implementations of these algorithms in a parallel spatial database system. 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 [OR93] can potentially be used for this purpose.

9 Acknowledgements

We would like to thank Joseph Burger for cheerfully fixing the hardware problems that we had on our network of PCs. The entire Paradise team has been very helpful and accommodating while we worked on this pa-per. We would like to thank both Intel and IBM for their generous hardware donations to the Paradise project. Funding for the Paradise project is provided by NASA under contracts #USRA-5555-17, #NAGW-3895, and #NAGW-4229 and ARPA through ARPA Order number 018 monitored by the U.S. Army Re-search Laboratory under contract DAAB07-92-C-Q508. Biswadeep Nag's critical analysis of earlier drafts of this paper has helped improve this presentation.

References

[AOT+95] D.J. 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.

[BKS93] 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.

[BKS96] 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, February 1996. IEEE Computer Society.

[BKSS94] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. "Multi–step Processing of Spatial Joins". In *Proceedings of the 1994 ACM-SIGMOD Conference*, Minneapolis, May 1994.

[DCW92] *"VPFView 1.0 Users Manual for the Digital Chart of the World"*. Defense Mapping Agency, July 1992.

[DG92] D. J. DeWitt and Jim Gray. "Parallel Database Systems: The Future of Database Processing or a Passing Fad?". *Communication of the ACM*, June, 1992.

[GS87] R. H. Güting and W. Shilling. "A Practical Divide–and–Conquer Algorithm for the Rectangle Intersection Problem". In *Information Sciences*, volume 42, 1987.

[Gün93] O. Günther. "Efficient Computation of Spatial Joins". In *IEEE Transactions on Knowledge and Data Engineering*, 1993.

[Gut84] A. Gutman. "R-trees: A Dynamic Index Structure for Spatial Searching". In *Proceedings of the 1984 ACM-SIGMOD Conference*, Boston, Mass, June 1984.

[HJR97] 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, August 1997.

[HS94] E. G. Hoel and H. Samet. "Performance of Data-Parallel Spatial Operations". In *Proceedings of the 20th VLDB Conf.*, pages 156–167, Santiago, Chile, September 1994.

[HS95] E. G. Hoel and H. Samet. "Benchmarking Spatial Join Operations with Spatial Output". In *Proceedings of the 21st VLDB Conf.*, Zurich, Switzerland, September 1995.

[KF92] I. Kamel and C. Faloutsos. "Parallel R–Trees". In *Proceedings of the 1992 ACM-SIGMOD Conference*, San Diego, California, June 1992.

[KFK96] N. Koudas, C. Faloutsos, and I. Kamel. "Declustering Spatial Databases on a Multi—Computer Architecture". In *EDBT*, pages 592–614, 1996.

[KO90] 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.

[KS97] N. Koudas and K. C. Sevcik. "Size Separation Spatial Join". In *Proceedings of the 1997 ACM-SIGMOD Conference*, pages 324–335, Tucson, Arizona, USA, May 1997.

[LR94] M. L. Lo and C. V. Ravishankar. "Spatial Joins Using Seeded Trees". In *Proceedings of the 1994 ACM-SIGMOD Conference*, Minneapolis, May 1994.

[LR96] M. L. Lo and C. V. Ravishankar. "Spatial Hash–Joins". In *Proceedings of the 1996 ACM-SIGMOD Conference*, Montreal, Canada, June 1996.

[OR93] F. Olken and D. Rotem. "Sampling from Spatial Databases". In *Proc. of the 9th International Conference on Data Engineering*, pages 199–208, April 1993.

[Ore86] J. A. Orenstein. "Spatial Query Processing in an Object–Oriented Database System". In *Proceedings of the 1986 ACM-SIGMOD Conference*, 1986.

[PD96] J. M. Patel and D. J. DeWitt. Partition based spatial–merge join. In *Proceedings of ACM SIGMOD 1996*, pages 259–270, June 1996.

[PYK+97] 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.

[Qua97] *"Quantify User's Guide"*. Pure Software Inc., 1309 Spouth Mary Avenue, Sunnyvale CA 94087, 1997.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. "The SEQUOIA 2000 Storage Benchmark". In *Proceedings of the 1993 ACM-SIGMOD Conference*, Washington, D.C., May 1993.

[Spa99] Space imaging, product catalog. *"http://www.spaceimaging.com/"*, 1999.

[Ter99] Applications for spin-2 digital images and prints. *"http://www.terraserver.com/people.htm"*, 1999.

[TY95] K.-L. Tan and J. X. Yu. "A Performance Study of Declustering Strategies for Parallel Spatial Databases". In *The 6th International Conference on Database and Expert Systems Applications (DEXA)*, pages 157–166, London, United Kingdom, September 1995.

[WDJ91] C. B. Walton, A. G. Dale, and R.M. 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, September 1991.

[ZAT97] 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.

[ZG90] H. Zeller and J. Gray. "An Adaptive Hash Join Algorithm for Multiuser Environments". In *Proceedings of the 16th VLDB Conf.*, Brisbane, Australia, 1990.