Partition Based Spatial–Merge Join *

Jignesh M. Patel
Computer Sciences Department,
University of Wisconsin, Madison
jignesh@cs.wisc.edu

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

Abstract

This paper describes PBSM (Partition Based Spatial–Merge), a new algorithm for performing spatial join operation. This algorithm is especially effective when neither of the inputs to the join have an index on the joining attribute. Such a situation could arise if both inputs to the join are intermediate results in a complex query, or in a parallel environment where the inputs must be dynamically redistributed. The PBSM algorithm partitions the inputs into manageable chunks, and joins them using a computational geometry based plane–sweeping technique. This paper also presents a performance study comparing the the traditional indexed nested loops join algorithm, a spatial join algorithm based on joining spatial indices, and the PBSM algorithm. These comparisons are based on complete implementations of these algorithms in Paradise, a database system for handling GIS applications. Using real data sets, the performance study examines the behavior of these spatial join algorithms in a variety of situations, including the cases when both, one, or none of the inputs to the join have an suitable index. The study also examines the effect of clustering the join inputs on the performance of these join algorithms. The performance comparisons demonstrates the feasibility, and applicability of the PBSM join algorithm.

1 Introduction

With the increasing popularity of automated processes in fields like Earth Sciences, Cartography, Remote Sensing, Land Information Systems etc., and the rapid increase in the availability of data from a wide variety of sources like satellite images, mapping agencies, simulation outputs etc., the last decade has witnessed an increase in the demands for systems that can store, manage, and manipulate spatial data. Increasingly, a database system has been employed to meet these requirements. Examples of commercial database systems that have been used for these applications are ARC/INFO [Arc95], Intergraph's MGE [Cor95], and Illustra [Ube94]). Data stored in these *spatial database systems* includes simple geometric types like points, lines, polygons, and surfaces, and more complex types like swiss–cheese–polygons (which are polygons with holes) that are derived from the simpler geometric types. A spatial database system must support queries on these spatial objects efficiently. Spatial database users frequently need to combine two spatial inputs based on some spatial relationship between the objects in the two inputs. For example, *map overlap*, which requires combining two maps to produce a third, is an important operation in a spatial database [Bur86, MGR91]. This operation of combining two inputs based on their spatial relationship is called a *spatial join*. Spatial joins, just like their counterparts in a relational system, are an expensive operation. Consequently, efficient spatial join algorithms are a critical component of any spatial database system.

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

- **Filter Step:** In this step, an approximation of each spatial object, such as its minimum bounding rectangle, is used to eliminate tuples that cannot be part of the result. This step produces *candidates* that are a superset of the actual result. These candidates are usually represented as a pair of object identifiers.
- **Refinement Step:** In this step, each candidate is examined (which usually requires fetching a pair of objects from disk) to check if it is part of the result. This check generally requires running a CPU–intensive computational geometry algorithm.

Numerous algorithms have been proposed to execute the filter step of a spatial join. Many of the earlier algorithms are based on transforming an approximation of a spatial object into another domain (e.g. a 1–dimensional domain), and performing the filter step in the new domain [OM88, Ore86, BHF93]. The drawback of this approach is that in the new domain some spatial proximity information is lost, making the algorithms complex and less efficient. Most of the newer algorithms are based on using spatial indices for performing the filter step of the spatial join [BKS93, Gün93, HS95], and require a spatial index on both the join inputs. These *tree join* algorithms can be described as synchronized depth–first

* This work was partially supported by NASA Contracts #USRA–5555-17, #NAGW–3895, and #NAGW–4229, and by an IBM Research Initiation Grant.

searches of both indices, with the two depth–first searches being guided by hints from each other.

While spatial indices might exist on both the inputs during the join of two base relations, there are many situations where neither join input will have a spatial index built on it. Such a situation could arise if the inputs to the join are intermediate results in a complex query, or in a parallel environment where the inputs have been dynamically redistributed. A spatial DBMS must evaluate these joins efficiently. One solution to this problem is to build a spatial index on both inputs and then use a tree join algorithm [LR95]. Another solution to this problem comes from the VLSI domain where one needs to compute the pairwise intersection between two potentially large sets of rectangles that don't fit entirely in main memory [GS87]. However, the VLSI algorithms are generally not very efficient with respect to the number of disk I/Os.

This paper makes two contributions. First, it presents a new spatial join algorithm, called the Partition Based Spatial–Merge (PBSM) Join, that does not require indices on either of its inputs. The algorithm partitions the inputs into "manageable" chunks and joins the chunks using a computational geometry algorithm that can be considered as the spatial equivalent of sort–merge. The algorithm incorporates a complete solution to the spatial join problem as it performs both the filter and the refinement step.

Second, it includes the results of a comprehensive performance study of three spatial join algorithms: a simple indexed nested loops based join algorithm, an R–tree based join algorithm, and the PBSM algorithm. The performance study is based on actual implementations of the three algorithms in Paradise [DKL$^+$94], which is an experimental GIS database system. Using real data from the TIGER [Tig] and the Sequoia [SFGM93] data sets, the study examines the behavior of the algorithms in a variety of situations, including the cases when none, one, or both the inputs to the join have a suitable index. The study also investigates the effect of clustering the join inputs. Many of the tree–based join algorithms that have been considered in earlier performance studies, use multiple inserts to build an index [HS95, LR94]. It is a well known fact that bulk loading an index is much more efficient than performing multiple inserts to construct it. For example, using a buffer pool size of 16MB, Paradise takes 109.9 seconds to bulk load 122K objects into an 6.5MB R*-tree index, and 864.5 seconds to build the same index using multiple inserts! Hence, in this study, for both the indexed nested loops and the R–tree based join algorithms, whenever required, indices are built using the Paradise bulk loading mechanism.

The remainder of this paper is organized as follows. Section 2 summarizes the related work in this area. Section 3 describes the PBSM algorithm. The performance study comparing various spatial join algorithms is presented in Section 4. Finally, Section 5 contains our conclusions and some future plans.

2 Related Work

As mentioned in the introduction, spatial join algorithms operate in two steps: a filter step and a refinement step. Most of the spatial join algorithms that have been proposed previously only solve the filter step. In this section, we summarize relevant work in this area. Throughout this section, we use the term spatial join to refer to the filter step of the spatial join.

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

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

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

	Require Use of an Index	Operate without an Index
Transform the approximation into another dimension	• Z–values [OM84], • Join Indices [Rot91] • Grid Files [HNKT90, BHF93] • kd–trees [KHT89, HNKT90]	• Z–values [Ore86, OM88]
Use the approximation directly in the two dimensional space	• Synchronized Tree Traversal [BKS93, Gün93, HS95] • Build 1 or 2 indices before joining [LR94, LR95]	• External VLSI algo [GS87] • PBSM • Spatial Hash Join [LR96]

Table 1: Classification of Various Spatial Join Algorithms

data structures. In [HS95], Hoel and Samet propose a tree join algorithm for the PMR quad tree, and compare the efficiency of variants of the PMR quad tree with variants of the R–tree [HS95].

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

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

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

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

3 Partition Based Spatial–Merge Join

This section describes a new algorithm, called the Partition Based Spatial–Merge (PBSM) join, for evaluating a spatial join. For the sake of concreteness, let R and S denote the two inputs to the join. We assume that the inputs are a sequence of tuples, and that each tuple has a spatial attribute that is used in the join condition. We also assume that the system has a unique identifier for each tuple. This unique identifier is referred to as the *OID* of the tuple.

The PBSM algorithm operates in the following two steps.

- **Filter Step:** The spatial attribute involved in the join may be a complex spatial feature like a polygon, a polyline, or a swiss–cheese polygon. In this step, the PBSM algorithm makes use of an approximation of the spatial feature to get a "rough estimate" of the characteristics of the spatial attribute. The minimum bounding rectangle (*MBR*), is used as an approximation in this step. The filter step uses partitioning to partition large inputs into smaller chunks. A computational geometry plane–sweeping technique is used to join the chunks. The result of the filter step is a set of *OID pairs* such that one OID of the pair refers to a tuple from the input R and the other OID refers to a tuple from the input S. Furthermore, for each pair, the MBR of the spatial join attribute of the R tuple overlaps with the MBR of the spatial join attribute of the S tuple.
- **Refinement step:** Since two non overlapping spatial features can have overlapping $MBRs$, and since the filter step "joins" the inputs based on the MBR of the joining attributes, the filter step generally will produce a superset of the join result. In the *refinement step*, the R and S tuples represented by the *OID pair* produced by the previous step are fetched from disk, and their join attributes are examined to determine if the join predicate is actually satisfied.

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

3.1 Filter Step

The filter step of the PBSM algorithm, begins by reading the tuples from the input R. For each tuple of the input R, the MBR of the joining attribute and the OID of the tuple, which is collectively called a *key–pointer element*, are appended to a temporary relation on disk. Let us denote this relation by R^{kp}. Similarly, the input S is read and a relation S^{kp} is produced. The goal of the filter step is to "pair" tuples from R and S such that the $MBRs$ of their join attributes overlap. R^{kp} and S^{kp} have the $MBRs$ for the join attributes of both the inputs R and S. The problem then simplifies to finding all $MBRs$ in R^{kp} that intersect with some MBR in S^{kp}. Rectangle intersection (the $MBRs$ are rectangles) has been extensively studied in the computational geometry field [PS88]. Given two sets of rectangles, such that **both** the sets fit entirely in main memory, efficient computational geometry algorithms, based on *plane–sweeping* techniques, exist for reporting all pairs of intersecting rectangles between the two sets. Now, if both R^{kp} and S^{kp} fit in memory, then a plane–sweeping algorithm can be used to find all pairs of R^{kp} and S^{kp} key–pointer elements that have overlapping MBRs. For such "matching" key–pointer elements pairs, the *OID* information is extracted, and the OID pair is added to the output of this step.

If R^{kp} and S^{kp} are too large to fit entirely in memory, then each is divided into P (non–disjoint) partitions R_1^{kp}, R_2^{kp}, ... R_P^{kp} and S_1^{kp}, S_2^{kp}, ... S_P^{kp} respectively. These partitions are formed in a way such that for each key–pointer element in a partition R_i^{kp}, all the key–pointer elements of S^{kp} that have an overlapping MBR are present in the corresponding S_i^{kp} partition. Furthermore, the size of the partitions are such that for each i ($1 <= i <= P$) R_i^{kp} and S_i^{kp} can **both** fit simultaneously in memory.

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

- From the catalog information for the joining attribute of input R, the algorithm estimates the *universe* of the input. The universe for a particular spatial join attribute is the rectangle that is the minimum cover of the join attribute of all the tuples in the input.
- The universe is then decomposed into a P subparts (P is the number of partitions). As an example, consider Figure 1 where the number of partitions is 4.
- Finally, the spatial partitioning function (see Section 3.4 for more details) is applied to the *MBR* of a key–pointer element. The partitioning function determines all the subparts of the universe with which the *MBR* overlaps, and inserts the key–pointer element into each partition corresponding to these subparts. Thus, if a *MBR* overlaps with multiple subparts of the universe, then it is inserted into multiple partitions. For example, the key–pointer element for the object shown in Figure 1, will be inserted into partitions 0 and 2.

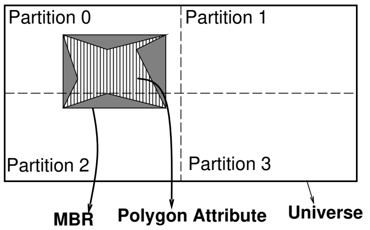

Figure 1: The Spatial Partitioning Function.

After both the inputs R and S have been partitioned, the algorithm joins the partitions using a computational geometry based plane–sweeping technique [PS88]. This technique, which was also used in [BKS93] for joining the entries of two R*–tree nodes, can be thought of as the spatial equivalent of the sort–merge algorithm. The details of the algorithm for merging the partitions R_i^{kp} and S_i^{kp} are as follows. Let $MBR.xl$ represent the lower x-coordinate of a MBR and let $MBR.xu$ represent the upper x-coordinate. First, the inputs R_i^{kp} and S_i^{kp}, which are a sequence of key–pointer elements, are sorted on the lower x values of the MBR, namely $MBR.xl$. Then, the $MBRs$ from the first key–pointer elements of R_i^{kp} and S_i^{kp} are examined, and the MBR which has a smaller $MBR.xl$ value is selected. Let r denote this MBR, and let us assume that r belongs to the input R_i^{kp}. Using the $r.xu$ value, the key–pointer elements of the input S_i^{kp} are scanned until a key–pointer element whose MBR has a $MBR.xl$ value greater than $r.xu$ is reached. Effectively, all the elements of S_i^{kp} that overlap with r along the x–axis are scanned. Each of these elements of S_i^{kp}, is then checked for overlap with r along the y–axis[1]. If an overlap exists, then the OID pair corresponding to the OIDs in the key–pointer elements is added to the result (the result of the filter step is a set of OID pairs). After this, r is marked as processed and is removed from consideration for the input R_i^{kp}. The algorithm continues by picking from the unprocessed part of the inputs R_i^{kp} and S_i^{kp}, the element that has the smallest $MBR.xl$ value. The smallest element is then "joined" with elements in the other input. This continues until one of the two inputs has been fully processed.

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

After joining each pair of partitions, the result is a relation whose tuples have the form $< OID_R, OID_S >$, such that the MBR of the joining attribute of the R tuple corresponding to OID_R overlaps with the MBR of the joining attribute of the S tuple corresponding to OID_S. Since the partitioning in the filter step might insert a tuple into multiple partitions, there could be duplicates in this relation. The refinement step eliminates these duplicates, and examines the actual R and S tuples to determine if the attributes actually satisfy the join condition. To avoid random seeks in fetching the R and S tuples, a strategy similar to that used in [Val87] is employed. First, the OID pairs are sorted using OID_R as the primary sort key and OID_S as the secondary sort key. Duplicates entries are eliminated during this sort. Next, as many R tuples as can fit in memory are read from disk along with the corresponding array of $< OID_R, OID_S >$ pairs. The OID_R part of this array is "swizzled" to point to the R tuples in memory, and then the array is sorted on OID_S (this makes the accesses to S sequential). The S tuples are then read sequentially into memory, and the join attributes of the R and the S tuple are checked to determine whether they satisfy the join condition.

[1] This check for overlap can be speeded up by organizing the $MBRs$ of S_i^{kp} that overlap with r along the x–axis in an interval–tree [PS88].

3.3 Determining the Number of Partitions

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

$$P = \lceil \frac{(||R|| + ||S||) * Size_{key-ptr}}{M} \rceil \tag{1}$$

3.4 Choosing a Spatial Partitioning Function

We now explore some of the alternatives that exist in selecting a spatial partitioning function. The spatial partitioning function described in Section 3.1 decomposes the universe into P subparts (where P is the number of partitions determined by Equation 1). However, in the presence of a non–uniform distribution, this partitioning function could produce partitions that have large differences in their sizes. As an example, consider Figure 2 where the number of partitions is 4. Here, most of the tuples are in the top left corner, and the spatial partitioning function will map all these tuples to partition 0. Partitions 2 and 3, on the other hand will have very few tuples.

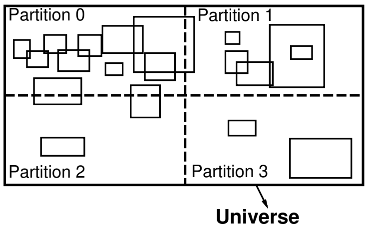

Figure 2: Spatial Partitioning Skew

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

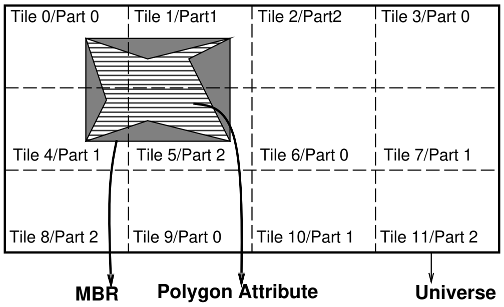

Figure 3: Spatial Partitioning Function using Tiles.

The spatial partitioning function just described is the spatial analog of virtual processor round robin partitioning for handling skews in parallel relational joins [DNSS92]. A similar partitioning function has been independently proposed for redundancy–based declustering of spatial objects in a parallel spatial database [TY95], but in that proposal the number of tiles always equals the number of partitions.

The design space for choosing the spatial partitioning function has two axes: the number of tiles used in the partitioning function, and the tile–to–partition mapping scheme. Decomposing the universe into small tiles produces many small containers, which are easier to pack into partitions to produce a more uniform partition distribution. However, spatial objects that span tiles from multiple partitions have to be replicated in all the partitions, thereby increasing the replication overhead. For the tile–to–partition mapping scheme, one could use either round robin or hashing on the tile number.

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

First, we explore the design space of the spatial partitioning function using the Tiger data. Figure 4 shows the effect of increasing the number of tiles, and choosing different tile–to–partition mapping schemes. The graph uses the coefficient of variation[2] of the distribution of the tuples in each partition as its metric. A perfect spatial partitioning function would be one that assigns equal number of tuples to each partition, and, consequently, would have a coefficient of variation of 0 for the distribution of the tuples in each partition. From Figure 4, the following observations can be made. First, using a large number of tiles and hashing on the tile number gives a good partitioning function. Second, all the partitioning functions improve as the number of tiles is increased. This is because with a larger number of tiles, "dense" regions get subdivided into more tiles, and these tiles can be mapped to different partitions. Third, for a given number of tiles, the partitioning function yields a more uniform distribution for

[2] The coefficient of variation is defined as the standard deviation divided by the mean.

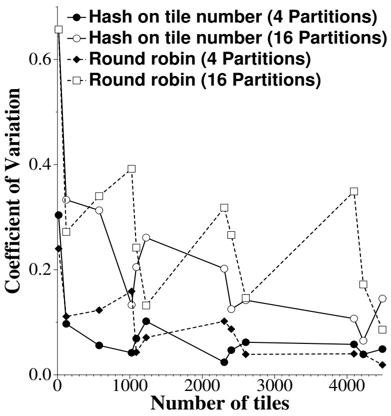

Figure 4: Spatial Partitioning Function Alternatives: Tiger Road Data

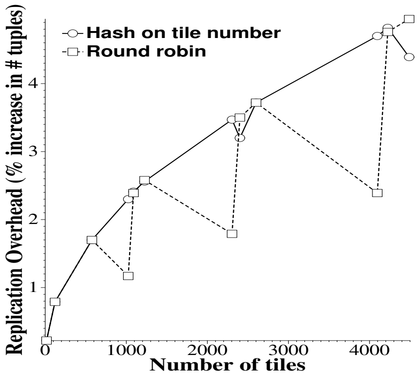

Figure 5: Replication Overhead: Tiger Road Data (16 Partitions).

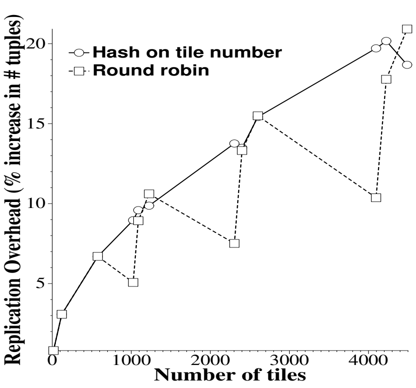

Figure 6: Replication Overhead: Sequoia Polygon Data(16 Partitions).

a smaller number of partitions (compare the graphs for hashing with 4 and 16 partitions). This, once again, is because the distribution of tiles that cover "dense" regions is better with a smaller number of partitions. For example, when the universe is decomposed into 25 tiles, 13 tiles cover 81.5% of the data. These 13 tiles can be spread across 4 partitions more uniformly than over 16 partitions, and, as a result, the coefficient of variation for 4 partitions is lower than that for 8 partitions.

Figure 5 measures the replication overhead—the increase in the number of tuples created due to replication during partitioning—for various number of tiles. The Figure shows that, for the Tiger data set, the replication overhead is very modest even for a very large number of tiles (increasing the number of tuples by 4.8% for 4000 tiles). The figure also shows some spikes in the curve for round robin. This is because with round robin, when the number of tiles is an integral multiple of the number of partitions, it is possible for an entire column to get mapped to a single partition. This is equivalent to having fewer number of tiles (each column now behaves like a single tile), Consequently, fewer tuples have to be replicated. However, at these points, the partitions produced by the partitioning function are less balanced (observe the jumps in Figure 4 for round robin with 16 nodes).

For the Sequoia data, we found that the effect of increasing the number of tiles on the tile–to–partition mapping scheme is very similar to the effect on the Tiger Data. However, the replication overhead, which is shown in Figure 6, is much higher.

3.5 Handling Partition Skew

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

4 Performance Evaluation

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

The remainder of this section is organized as follows. First the index nested loops and the R–tree based join algorithms are described. This is followed by a description of the methodology used in the study, and, finally, the results of the study are presented.

4.1 Indexed Nested Loops Join

Let R and S denote the two relations that are being joined, and assume that R has fewer tuples than S. If neither join input has an index on the joining attribute, the indexed nested loops join algorithm first builds an index on the smaller input R. The index is built using a bulk loading mechanism that reads the extent R and extracts the *key–pointer information* ($< MBR, OID >$) for each tuple. The key–pointer information is then spatially sorted based on the MBR. Spatial sorting is accomplished by transforming the center point of

Data Type	# of Objects	Total Size	R*–tree Size
Road	456,613	62.4 MB	24.0 MB
Hydrography	122,149	25.2 MB	6.5 MB
Rail	16,844	2.4 MB	1.0 MB

Table 2: Wisconsin TIGER Data

Data Type	# of Objects	Total Size	R*–tree Size
Polygons	58,115	21.9 MB	3.0 MB
Islands	21,007	6.2 MB	1.1 MB

Table 3: Sequoia Data.

the MBR into a Hilbert value, and using this value for ordering the key–pointer information. This sorting brings together key–pointers whose joining attributes are spatially close. The spatial index, which in our case is a R*–tree, is then built in a bottom up fashion [DKL+94]. After building the index on the join attribute of R, a scan is started on S. Each tuple of S is used to probe the index on R. The result of the probe is a set of (possibly empty) $OIDs$ of R. The tuples of R corresponding to these $OIDs$ are then fetched (from disk, if necessary) and checked with the S tuple to determine if the join condition is satisfied. Fetching each R tuple from disk will generally incur a random disk I/O.

4.2 R–tree Based Join Algorithm

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

The R–tree join algorithm of [BKS93] only performs the filter step of the spatial join, and produces a set of candidate OID pairs corresponding to the objects whose MBRs intersect. The objects corresponding to these OIDs then have to be fetched and checked to determine if the join predicate is actually satisfied. For this, we use the same technique that was used in the PBSM join algorithm (refer to Section 3.2).

4.3 Methodology

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

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

For the performance study, the PBSM algorithm used 1024 tiles for the spatial partitioning function. We explored the effect of the number of tiles on the execution time of PBSM, but found that changing the number of tiles had a very small effect on the overall execution time (less than 5%). The full length version of this paper [PD] presents this result.

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

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

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

For the third collection, the islands and polygon data sets from the Sequoia 2000 Storage Benchmark [SFGM93] were used. The polygon data set represents regions of homogeneous landuse characteristics in the State of California and Nevada, while the island data set represents holes in the polygon data (example, a lake in a park). The average number of points in a polygon tuple is 46, and the average number of

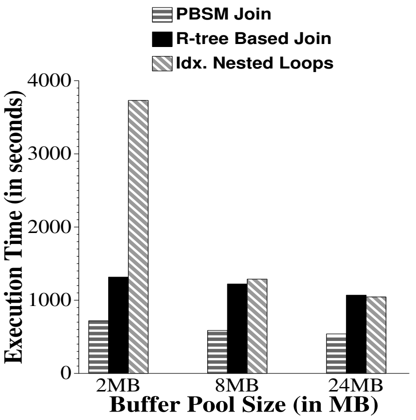

Figure 7: TIGER Data: Join Road with Hydrography

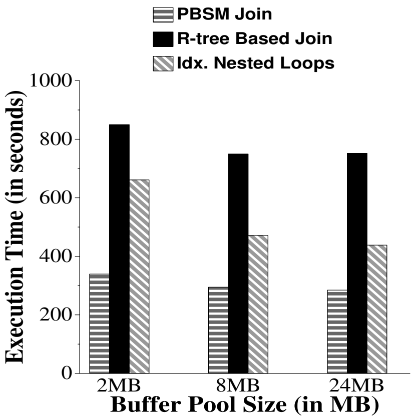

Figure 8: TIGER Data: Join Road with Rail

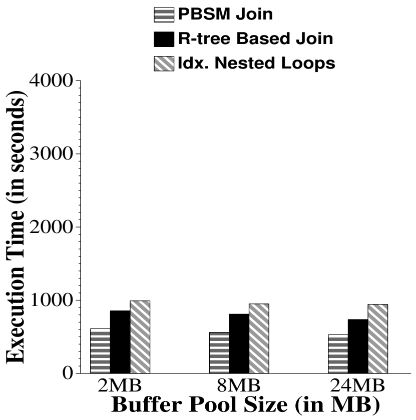

Figure 9: Clustered TIGER Data: Join Road with Hydrography

points in an islands tuple is 35. The query used in this experiment joined the polygons and islands, producing, as a result, those islands that are contained in one or more of the polygons. This result contained 25,260 tuples and was 30.8MB in size. The characteristics of this data are shown in Table 3.

4.4 Joins When None of the Indices Pre–exist

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

Comparison Using the TIGER Data

In the first experiment, the Road data set was joined with the Hydrography data set. Figure 7 shows the execution times of the three spatial join algorithms as a function of the buffer pool size. For this query, the PBSM algorithm is 80–100% faster than the R–tree based join algorithm, and 93–400% faster than the Indexed Nested Loops algorithm. The Indexed Nested Loops builds an index on Hydrography and probes it with the tuples of the Road data set. After probing the index with a tuple, the Indexed Nested Loops Join fetches the matching Hydrography tuple, and examines it to determine if the join predicate is actually satisfied. For smaller buffer pool sizes, fetching the matching Hydrography tuple generally requires a disk I/O. However, as the size of the buffer pool increases, larger portions of the Hydrography data reside in the buffer pool, and, as a result, the performance of the Indexed Nested Loops Join improves significantly.

Next, the performance of the algorithms was compared by joining the Road data with the Rail data (Figure 8). Since the size of the Rail data is only 2.4MB, and the index on it is only 1.0MB, the index and the data usually fit in the buffer pool (the Rail data pages compete with the Road data pages for buffer pool frames). As a result, the Indexed Nested Loops performs better than the R–tree based join algorithm. The cost of the R–tree based join algorithm is dominated by the cost of building the index on the larger Road data, and the algorithm spends about 85% of its total time building this index.

Effect of Clustering

Next, to investigate the effect of clustering, we ran the query that joins Roads with Hydrography on the clustered TIGER data (clustering had a similar effect on the join of Road with Rail, and these results can be found in [PD]). The results of this experiment are shown in Figure 9. For this experiment, PBSM is about 40% faster than the R–tree based join algorithm, and 60–80% faster than Indexed Nested Loops.

By comparing Figures 9 and 7, one can observe that if the inputs to the join are clustered, the performance of all the join algorithms improve. To understand this behavior, consider Figures 10 through 12, which contain detailed cost breakdowns for both the clustered and the non–clustered scenarios. For the R–tree based join algorithm, the total join cost includes the cost of building both indices, the cost of joining the indices, and the cost of fetching the Road and Hydrography tuples from the disk and examining them to determine if the join predicate is actually satisfied (the refinement step). For the Indexed Nested Loops join, the total cost consists of building the index on Hydrography, and then probing it repeatedly with tuples from the Road data set. For the PBSM algorithm, the total join cost includes the cost of forming the two partitions, the cost of merging the partitions, and the cost of the refinement step.

First, consider the individual costs of the R–tree based join algorithm (Figure 10).

- **Index Building Costs:** The indices for this join are built using bulk loading (refer to Section 4.1 for a more detailed description of bulk loading). Bulk loading an index has three costs: the cost of extracting the key–pointers from the input, sorting the key–pointers, and building the index using the sorted key–pointers. When an input is clustered, sorting the key–pointers can be avoided, thereby, reducing the cost of building the index.

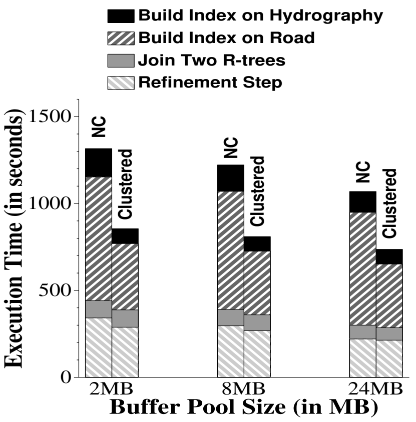

Figure 10: R–tree Based Algorithm, TIGER Data: Join Road with Hyd. Clustered and non–clustered (NC) scenario.

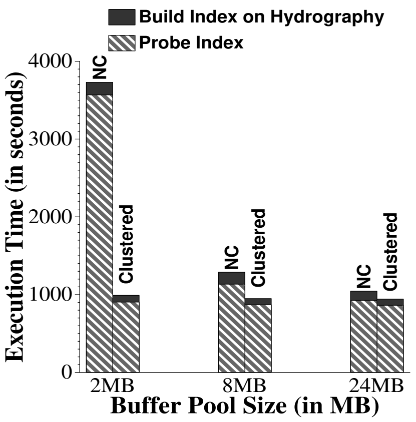

Figure 11: Indexed Nested Loops Algorithm, TIGER Data: Join Road with Hyd. Clustered and non–clustered (NC) scenario.

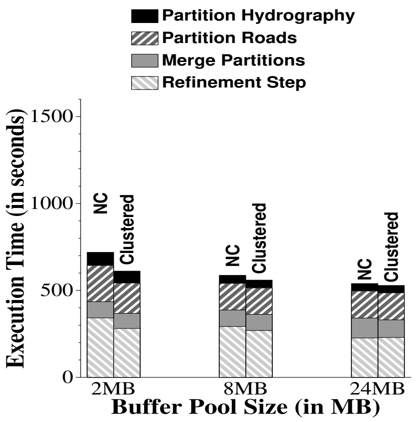

Figure 12: PBSM Algorithm, TIGER Data: Join Road with Hyd. Clustered and non–clustered (NC) scenario.

- **Tree Joining Cost:** Since bulk loading sorts the keys in the non–clustered scenario, the trees that are built in both the clustered and the non–clustered scenarios are exactly the same. Consequently, the algorithm for joining the two trees performs exactly the same steps in both the cases, and, as a result, clustering has no effect on the time for joining the indices.
- **Refinement Step Cost:** In the refinement step, tuples of R are scanned once, and the tuples of S are scanned multiple times. The refinement step (refer to Section 3.2) reads a bunch of R tuples that are physically clustered on the disk, and then reads the S tuples that "spatially match" these R tuples. When the physical order of the tuples on disk is the same as the spatial order, the fetches to the S tuples scan only a small portion of the relation. Consequently, the refinement costs improve with spatial clustering.

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

Now, consider the Indexed Nested Loops Join (Figure 11). Clustering has a similar effect on the index building cost, as sorting the keys can be avoided. Further, for small buffer pool sizes the index on Hydrography cannot fit entirely in memory, and the index probe cost is significantly reduced when the data is clustered. This effect is similar to the behavior of the indexed nested loops join in the relational domain where sorting the relation that is used to probe the index improves the performance of the join.

Besides a reduction in the refinement cost, clustering reduces the partitioning costs of the PBSM algorithm (Figure 12). The difference in the partitioning costs is more pronounced for smaller buffer pool sizes. This behavior is due to the way partitions are written out to disk. The PBSM algorithm does not manage any of the partition buffers; it simply writes tuples to appropriate partition files, and relies on the SHORE storage manager to flush pages of the partition files to disk. The spatial partitioning function decomposes the universe into tiles (refer to Section 3.4) and maps the tiles to partitions. When the input is clustered, consecutive tuples are more likely to be in the same tile, and, as a result, get mapped to the same partition. Consequently, this phase incurs at most one disk seek for each tile. When the data is not clustered, the partitions fill up in a random order, and, as a result, writing the partitions now involve many disk seeks over the partition files.

An interesting point to note from Figures 10 and 12, is that the PBSM and the R–tree based join algorithm have the same elapsed time for performing the refinement step. For PBSM, the refinement cost constitutes about 45% of the overall join cost, and for the R–tree based algorithm, the refinement cost is about 23% of the overall join cost. For performing the refinement step, which in this case requires examining two polylines for intersection, a plane–sweeping algorithm was used. Without this, the cost of the refinement step increases by 62%.

Comparison Using the Sequoia Data

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

For the Sequoia data set, we observed that the cost of the refinement step is a dominating factor for both the PBSM and the R–tree based join, contributing about 79% to the overall PBSM join cost, and 68% to the overall R–tree based join cost. (The detailed cost break for this data set appears in the full length version of this paper [PD]). The refinement step for this query involves checking if an island polygon is contained in a landuse polygon. This check for containment is currently implemented in Paradise using a naive $O(n^2)$ algorithm (n is the number of points in a polygon). There are a number of techniques for reducing the cost of this part of

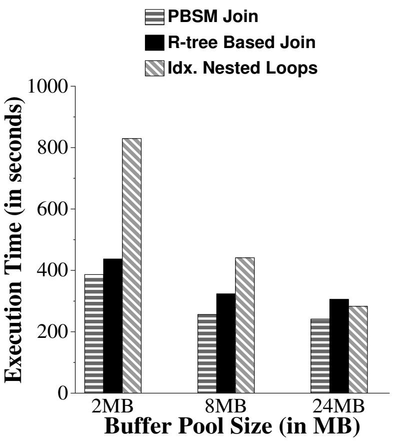

Figure 13: Sequoia Data Set.

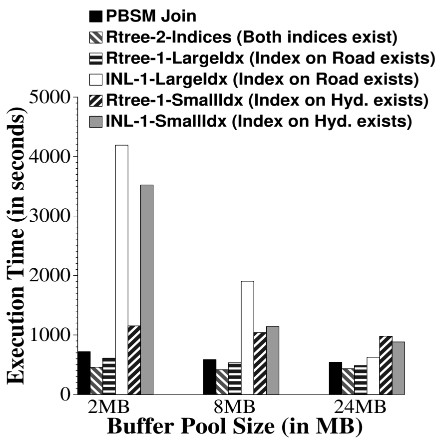

Figure 14: Comparison of the Join Algorithms with indices, TIGER Data (Join Road with Hydrography).

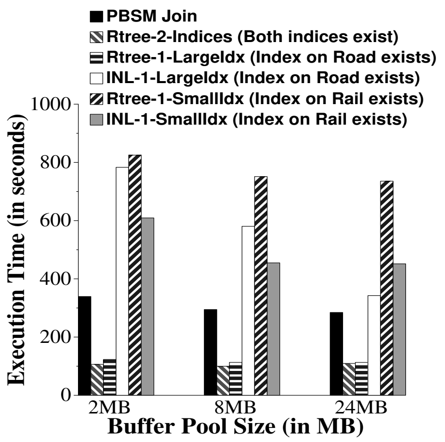

Figure 15: Comparison of the Join Algorithms with indices, TIGER Data (Join Road with Rail).

the join [BKSS94] (by an order of magnitude in many cases). These techniques rely on using as a filter in the refinement step, extra information that is precomputed and stored along with each spatial feature. As an example, each polygon could store its minimum bounding rectangle (MBR), and a maximal enclosed rectangle (MER)—which is a rectangle that is fully contained in the polygon. Then, during the refinement step to determine if polygon p1 is contained in polygon p2, the MBR of p1 could be examined for containment in the MER of p2. If this containment holds, p1 is guaranteed to lie within p2, and we can skip further processing. If these techniques were implemented, the relative performance of the PBSM algorithm would improve.

Summary of the No Pre–existing Index Case

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

4.5 Joins in the Presence of Pre–existing Indices

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

The results of this experiment are shown in Figures 14 and 15. When indices pre–exist on both the inputs, the R–tree based algorithm has the best performance. Since building an index on the smaller input is not very expensive, the R–tree based algorithm also has the best performance when an index exists only on the larger input (in the graphs, compare PBSM, Rtree–1–LargeIdx and INL–1–LargeIdx). When an index exists only on the larger input, the Indexed Nested Loops encounters many buffer misses while probing the index. Even if we chose to build an index on the smaller input and probe it, the index probing cost itself is still greater than the R–tree join cost (compare Rtree–1–LargeIdx and INL–1–SmallIdx).

In the last case, when an index exists only on the smaller input, the PBSM join performs better than the R–tree and the Indexed Nested Loops based joins (in the Figures 14 and 15 compare PBSM with Rtree–1–SmallIdx and INL–1–SmallIdx). For small buffer pool sizes, when joining Hydrography with Roads (Figure 14), the R–tree based algorithm (labeled as Rtree–1–SmallIdx) performs better than Indexed Nested Loops (labeled as INL–1–SmallIdx). However, as the buffer pool size increases with respect to the index size, the performance of Indexed Nested Loops improves rapidly, outperforming the R–tree based join for large buffer pool sizes in Figure 14, and for all buffer pool sizes in Figure 15.

The performance of the algorithms using the clustered TIGER data qualitatively matched with the results for the non–clustered case, while the performance of the different algorithms using the Sequoia data qualitatively matched the results shown in Figure 14. These numbers are omitted from this paper.

In summary, if an index exists on the larger input, or if both inputs have a pre–existing index, then the R–tree based join algorithm has the best performance, and if an index exists only on the smaller input, then the PBSM algorithm has the better performance.

4.6 CPU Costs

We now examine the CPU and the I/O costs involved in the spatial join algorithms. Table 4, shows the I/O costs incurred when joining the Road and the Hydrography data from the TIGER data set. Note that, except for the first component in each join algorithm, every component starts

Algo-rithm	Component of the Algorithm	24MB Buffer Pool			8MB Buffer Pool			2MB Buffer Pool		
		Total Cost	**I/O Cost**	**I/O Con–tribution**	**Total Cost**	**I/O Cost**	**I/O Con–tribution**	**Total Cost**	**I/O Cost**	**I/O Con–tribution**
PBSM	Partition Road	155.6	35.4	22.7%	153.1	38.0	24.8%	208.2	83.4	40.0%
	Partition Hyd	42.5	15.5	36.5%	46.2	14.3	30.9%	75.6	29.8	39.4 %
	Merge Partitions	114.6	16.3	14.2%	94.9	19.0	20.0%	93.4	20.4	21.8%
	Refinement	226.3	62.8	27.8%	297.4	99.7	33.5%	341.9	147.1	43.0%
	TOTAL	539.0	130.0	**24.1%**	591.6	171.0	**28.9%**	719.1	280.7	**39.0%**
R-Tree Join	Build Hyd. Index	125.0	11.4	9.1%	154.0	27.4	17.8%	163.1	31.2	19.1%
	Build Road Index	649.5	118.8	18.3%	679.6	110.3	16.2%	711.6	134.1	18.8%
	Join Indices	74.0	32.3	43.6%	91.7	44.5	48.5%	99.4	50.0	50.3%
	Refinement	220.5	64.1	29.1%	296.4	93.9	31.7%	341.7	136.4	39.9%
	TOTAL	1069.0	226.6	**21.2%**	1221.7	276.1	**22.6%**	1315.8	351.7	**26.7%**
NL-Idx	Build Hyd. Index	119.7	12.4	10.4%	150.9	28.9	19.2%	162.0	35.8	22.1%
	Probe Index	925.0	120.7	13.0%	1137.3	341.8	30.0%	3568.5	2369.1	66.4%
	TOTAL	1044.7	133.1	**12.7%**	1288.2	370.7	**28.8%**	3730.5	2404.9	**64.5%**

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

out with some dirty pages left behind in the buffer pool by the previous component. The Table shows that, for all the algorithms, the CPU costs dominate the I/O costs (by a large amount in most cases). The reason for this is two folds. First, performing spatial operations like probing an R*-tree index, joining partitions using a plane–sweep algorithm, and spatial sorting during bulk loading an index, are computationally intensive. Second, the SHORE storage manages works hard at minimizing the I/O costs. Whenever a dirty page has to be flushed to the disk, the storage manager forms a sorted list of all the dirty pages in the buffer pool, and tries to find pages that are consecutive on the disk. These pages are then written to the disk.

CPU costs were found to be a dominating factor for the spatial joins on the clustered TIGER and the Sequoia data sets too [PD]. Once again, due to space constraints we have omitted these graphs from this paper.

5 Conclusions and Future Work

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

This paper also contains the results of a comprehensive performance study that is based on actual implementation of three spatial join algorithms in Paradise, a database system for handling GIS applications. The three algorithms are: the traditional indexed nested loops algorithm, a previously proposed algorithm that uses spatial indices on both the inputs to evaluate the join, and the PBSM algorithm. The performance comparison, using real data sets, show that the PBSM algorithm is more efficient when neither of the inputs to the join have a spatial index. When an index exists only on the smaller input, the PBSM algorithm still performs better than the other algorithms. The R–tree based algorithm has better performance, when an index exists on the larger input, or if both inputs have a pre–existing index.

As part of our future work, we plan on investigating the use of parallelism to evaluate spatial joins. Since, PBSM, just like hash based relational joins, uses partitioning to break large inputs into smaller parts, we expect that the PBSM algorithm will parallelize efficiently. Parallelizing PBSM, would require a strategy for declustering spatial objects. The spatial partitioning function that is used by PBSM for partitioning large inputs, can also be used for declustering spatial data. We are currently examining these issues in the broader context of extending Paradise [DKL+94] to run on shared–nothing architectures [Sto86]. Parallel spatial databases are emerging as an attractive solution for storing and manipulating large volumes of spatial data [DLPY93], and some techniques for declustering spatial data have recently been proposed [TY95]. However, unless the spatial data is uniformly distributed, these techniques can result in unbalanced partitions. We feel that our spatial partitioning function using tiling, which is the spatial equivalent of virtual processor partitioning in a parallel relational system [DNSS92], would probably adapt better to different data distributions. We are also evaluating various tradeoffs in declustering spatial data. Since the spatial partitioning function might map an input object to multiple outputs, one could either replicate such objects entirely, or replicate just the spatial approximation (like the minimum bounding rectangle). If the object is not replicated in its entirety (as in [TY95]), then remote fetches might be required, whereas if the object is fully replicated, remote fetches can be avoided at the expense of an increase in the amount of storage. The tradeoff for many of these would probably depend on the characteristics of the input data and the queries on them.

6 Acknowledgement

We would like to thank Jie–Bing Yu for patiently answering our questions about Shore. We would also like to thank Navin Kabra, Praveen Seshadri, Joe Hellerstein and the SIGMOD referees for provided useful feedback on earlier drafts of this paper.

References

[Arc95] ESRI, Redlands, CA. *"ARC/INFO: The World's GIS. An ESRI White Paper"*, March 1995.

[Ben75] J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In *Communication of the ACM*, volume 18(9), September 1975.

[Ben79] J. L. Bentley. "Multidimensional Binary Search Trees in Database Applications". In *IEEE Transactions on Software Engineering*, volume 5(4), 1979.

[BHF93] L. Becker, K. Hinrichs, and U. Finke. "A New Algorithm for Computing Joins With Grid Files". In *IEEE Transactions on Knowledge and Data Engineering*, 1993.

[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.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles". In *Proceedings of the 1990 ACM-SIGMOD Conference*, June 1990.

[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.

[Bur86] P. A. Burrough. *"Principles of Geographic Information Systems for Land Resources Assessment"*. Oxford University Press, 1986.

[CDF+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. Tsatalos, S. White, and M. J. Zwilling. "Shoring up Persistent Applications". In *Proceedings of the 1994 ACM-SIGMOD Conference*, "Minneapolis, Minnesota", May 1994.

[CFR87] T. Sellis C. Faloutsos and N. Roussopoulos. "Analysis of Object Oriented Spatial Access Methods". In *Proceedings of the 1987 ACM-SIGMOD Conference*, San Francisco, May 1987.

[Cor95] Intergraph Corporation. "GIS/AM/FM Information". *"http://www.intergraph.com/utilmap.shtml"*, 1995.

[DKL+94] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J. Yu. "Client–Server Paradise". In *Proceedings of the 20th VLDB Conf.*, Santiago, Chile, September 1994.

[DLPY93] D. J. DeWitt, J. Luo, J. M. Patel, and J. Yu. "Paradise – A Parallel Geographic Information System". In *Proceedings of the ACM Workshop on Advances in Geographic Information Systems*, Arlington, Virginia, November 1993.

[DNSS92] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. "Practical Skew Handling in Parallel Joins". In *Proceedings of the 19th VLDB Conf.*, August 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.

[HNKT90] L. Harada, M. Nakano, M. Kitsuregawa, and M. Takagi. "Query Processing Methods for Multi–Attribute Clustered Relations". In *Proceedings of the 16th VLDB Conf.*, Brisbane, Australia, 1990.

[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.

[KHT89] M. Kitsuregawa, L. Harada, and M. Takagi. "Join Strategies on KD-Tree Indexed Relations". In *IEEE Transactions on Knowledge and Data Engineering*, 1989.

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

[LR95] M. L. Lo and C. V. Ravishankar. "Generating Seeded Trees From Data Sets". In *Proceedings of the Fourth International Symposium on Large Spatial Databases*, Portland, ME, August 1995.

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

[MC80] C. Mead and L. Conway. *"Introduction to VLSI Systems"*. Addison–Wesley, Reading, Mass., 1980.

[MGR91] D. J. Maguire, M. F. Goodchild, and D. W. Rhind. *"Geographic Information Systems"*, volume 1. Longman Scientific & Technical, copublished in the US with John Wiley & Sons, Inc. New York, 1991.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. "The Grid File: An Adaptable, Symmetric Multikey File Structure". *ACM Transactions on Database Systems*, March 1984.

[NS86] R. C. Nelson and H. Samet. "A Consistent Hierarchical Representation for Vector Data". In *Computer Graphics*, volume 20(4), August 1986.

[OM84] J. A. Orenstein and T. H. Merrett. "A Class of Data Structures for Associative Searching". In *Proceedings of the 3rd ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, 1984.

[OM88] J. A. Orenstein and F. A. Manola. "PROBE Spatial Data Modeling and Query Processing in an Image Database Application". In *IEEE Transactions on Software Engineering*, volume 14(5), May 1988.

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

[Ore89] J. A. Orenstein. "Redundancy in Spatial Databases". In *Proceedings of the 1989 ACM-SIGMOD Conference*, 1989.

[Ore90] J. A. Orenstein. "A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces". In *Proceedings of the 1990 ACM-SIGMOD Conference*, 1990.

[PD] J. M. Patel and D. J. DeWitt. "Partition Based Spatial–Merge Join". http://www.cs.wisc.edu/paradise/paradise.papers.html.

[PS88] F. P. Preparata and M. I. Shamos, editors. *"Computational Geometry"*. Springer, 1988.

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

[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.

[Sto86] M. Stonebraker. "The Case for Shared Nothing". *Database Engineering*, 9(1), 1986.

[Tig] U. S. Bureau of the Census, Washington, DC. *"TIGER/Line Files(TM), 1992 Technical Documentation"*.

[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)*, London, United Kingdom, September 1995.

[Ube94] M. Ubell. "The Montage Extensible DataBlade Architecture". In *Proceedings of the 1994 ACM-SIGMOD Conference*, May 1994.

[Val87] P. Valduriez. "Join Indices". In *ACM TODS*, volume 12(2), 1987.

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