Spatial Hash-Joins*

Ming-Ling Lo
Department of EECS
University of Michigan–Ann Arbor
1301 Beal Avenue, Ann Arbor, MI 48109
mingling@eecs.umich.edu

Chinya V. Ravishankar
Department of EECS
University of Michigan–Ann Arbor
1301 Beal Avenue, Ann Arbor, MI 48109
ravi@eecs.umich.edu

Abstract

We examine how to apply the hash-join paradigm to spatial joins, and define a new framework for spatial hash-joins. Our spatial partition functions have two components: a set of bucket extents and an assignment function, which may map a data item into multiple buckets. Furthermore, the partition functions for the two input datasets may be different.

We have designed and tested a spatial hash-join method based on this framework. The partition function for the inner dataset is initialized by sampling the dataset, and evolves as data are inserted. The partition function for the outer dataset is immutable, but may replicate a data item from the outer dataset into multiple buckets. The method mirrors relational hash-joins in other aspects. Our method needs no pre-computed indices. It is therefore applicable to a wide range of spatial joins.

Our experiments show that our method outperforms current spatial join algorithms based on tree matching by a wide margin. Further, its performance is superior even when the tree-based methods have pre-computed indices. This makes the spatial hash-join method highly competitive both when the input datasets are dynamically generated and when the datasets have pre-computed indices.

1 Introduction

Relational hash joins [1, 2, 3, 4, 5, 6, 7] yield excellent performance, particularly for relations that are large compared to buffer sizes. The hash-join paradigm is well-studied in the relational domain. However, due to difficulties peculiar to spatial joins, this method has not been directly applicable to the spatial domain. Existing spatial join methods have been tree-based for the most part, with *seeded trees* [8, 9] being a particularly efficient approach.

*This work was supported in part by the Consortium for International Earth Science Information Networking.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

The hash-join paradigm may yield other advantages in the spatial domain besides efficiency. First, existing hash-join software may be easily modified to implement a spatial hash-join, facilitating integration with existing database systems. Second, relational hash-join costs depend mainly on input data sizes and can be easily estimated. A spatial join following this paradigm may also lead to greater predictability in costs, facilitating spatial query planning and optimization. In contrast, the costs of spatial joins based on tree matching or other spatial indices can depend mostly on characteristics such as spatial distribution or ordering in input stream of the input data, and thus harder to estimate.

In this paper, we discuss the difficulties of applying the hash-join paradigm to spatial joins, and propose a framework for designing spatial hash-join methods. Based on this framework, we implement a spatial hash-join algorithm very similar to its relational counterpart, and with similar advantages. We evaluate our method by conducting experiments with a variety of data sets, including real-life data.

Like relational hash-joins, our method partitions its input into buckets during the *partition phase* and then joins the buckets to produce join results in the *join phase*. However, unlike relational joins, a partition function in our framework comprises two components: a set of *bucket extents* and an *assignment function*. The assignment function may map a data item into multiple buckets, and the partition functions for the two datasets may differ.

Spatial joins proceed in two stages: the filter step and the refinement step [10]. This paper focuses on the filter step. We view the refinement step as an important but orthogonal issue; any innovation in this step should also be usable with our method.

Our experiments show that our spatial hash-join method outperforms current spatial join algorithms based on tree matching by wide margins, even when the tree-based methods are given pre-computed indices. This makes our method highly competitive both when the input datasets are dynamically generated and when they have pre-computed indices.

This paper is organized as follows. Section 2 discusses related work. Section 3 studies the difficulties in applying the hash-join paradigm to spatial joins. Our framework for spatial hash joins is presented in Section 4. Section 5 presents our design of a spatial hash-join algorithm, and Section 6 presents its implementation. The results of our experiments are given in Section 7. Section 8 discusses related issues, and Section 9 concludes the paper.

2 Related Work

Previous methods for spatial joins have generally assumed pre-computed indices for the input datasets or other forms of pre-computation, with the seeded tree method [8, 9] being an exception. These algorithms include those based on join indices [11, 12], those based on z-ordering [13, 10, 14], and those based on tree-like indices [15]. Güting and Schilling [16] proposed a divide-and-conquer algorithm based on *seperational representation*. This method does not require spatial indices, but does require external sorting for large data sets.

Tree-based methods include those based on the R-tree and its variants [17, 18, 19, 20] and the seeded tree method [8, 9]. R-trees are commonly used as indices in spatial databases, and can be used to facilitate spatial joins when they exist. Brinkhoff *et al.* [21] proposed a method to join two existing R-trees, which consists of an R-tree matching algorithm and a collection of techniques to reduce CPU and disk I/O costs. The seeded-tree method [8, 9] may be used when no indices exist, and delivers better performance. A brief discussion of seeded trees also appears in Section 6.1.

The PBSM method of Patel and DeWitt [22] also requires no indices, and operates by first dividing the input into manageable partitions and joining them using plane-sweeping. This method is discussed further in Sections 4.3.1 and 8.

3 Difficulties with Spatial Joins

Two characteristics intrinsic to spatial joins prevent direct application of relational join algorithms to spatial joins. First, spatial data lack a natural total order preserving spatial closeness. Although techniques based on space-filling curves [13, 10, 14, 23] exist for defining total orders over sets of spatial objects, such total orders do not uniformly preserve spatial closeness, so two objects adjacent in space can sometimes be far away in the ordering. This difficulty prevents direct application of relational techniques to spatial databases. The second difficulty is that the relational hash-join method is optimized for handling equi-joins, but spatial join predicates are often much more complicated.

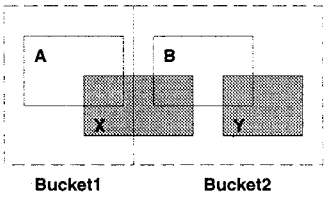

Figure 1: A simple spatial join. Data objects A and B belong to one dataset, and X and Y to another. Boxes with dashed lines represent hash buckets.

3.1 The Coherent-Assignment Problem

Relational hash-joins find pairs of tuples with identical join-attribute values. Their partition functions hash tuples with identical join-attribute values into the same bucket. Spatial joins, on the other hand, must identify pairs of objects with relationships more complicated than equality. Placing objects with identical spatial attributes into the same bucket does not suffice.

Difficulties arise at both the predicate and the join operator levels. Consider the spatial intersection join, which identifies pairs of objects whose spatial attribute values overlap. Consider three objects A, B and X, and denote the attribute value j of object A by $A.j$. For an equi-join predicate, if $X.j = A.j$ and $A.j \neq B.j$, we know that $X.j \neq B.j$. On the other hand, if we know that A intersects X, but A does not intersect B, we still know nothing about whether B and X intersect.

Suppose objects A and B belong to one dataset, and object X to another. In performing an equi-join on attribute j, we could hash tuples A and X into one bucket, and tuple B into another, and be certain there are no matching pairs across the two buckets. Unfortunately, we cannot do the same for a spatial intersection join. Figure 1 shows three pairs of matching objects (A, X), (B, X) and (B, Y). Although A and B do not intersect, X intersects both of them. If we hash A and X into one bucket, and B and Y into another, we will miss identifying the intersecting pair (B, X). Unless we hash all objects into one single bucket, we will always miss some intersecting pairs, no matter how we divide the objects between buckets.

An equi-join effectively constructs the equivalence classes induced by the equality predicate over values of the join attribute. The relational hash-join method works well because partition functions create buckets that fully include one or more of these equivalence classes. The property most crucial to the performance of the relational hash-join method is that the equi-join equivalence classes are not split across buckets. Thus, given a join predicate, we do not have to match objects across buckets. We term this a *coherent assignment* of attribute values to buckets with respect to a join predicate.

Unfortunately, spatial join predicates do not always

define such equivalence classes, and it is impossible to guarantee a coherent assignment of objects to hash buckets with respect to spatial join predicates. That is, we cannot divide the objects in the two datasets into n groups, and ensure that the two objects appearing in any matched pair always belong in the same group. We call this the *coherent-assignment problem*. This difficulty and the fact that spatial attributes are of higher dimensions, and thus more complicated, make spatial hashing a difficult problem.

4 Our Spatial Hash-Join Framework

We present our framework for designing spatial hash-join algorithms in this section. Hash-join algorithms for various spatial join predicates can be realized through appropriate choices for the design parameters of this framework. We borrow from relational-join terminology, and call the operand datasets of a spatial hash-join the *inner* and the *outer* datasets. Buckets produced by partitioning the inner (outer) dataset or relation are called the *inner (outer) buckets*. Our algorithmic framework, like that of the relational hash-join, has two phases: the partition phase and the join phase.

Partition phase: place inner and outer dataset objects into buckets using *spatial partition functions*.

Join phase: join corresponding inner and outer buckets to obtain results.

Relational hash-join algorithms have conformed to two principles, one relating to the partition phase, and the other to the join phase:

Single assignment: The partition function assigns each input data item to exactly one bucket.

Single matching: Each bucket of the inner relation is joined with exactly one bucket of the outer relation.

The partition phase divides the inner and outer relations into some number of bucket pairs, each pair having one inner bucket and one "corresponding" outer bucket. The "single matching" principle states that each inner or outer bucket appears in exactly one bucket pair. We will call a pair of buckets to be joined in the join phase a *join bucket pair*.

4.1 Primary Design Alternatives

The coherent assignment problem requires us to relax the single-assignment or single-matching principle. The two main alternatives are:

Multiple assignment: The partition function may map an input object into multiple buckets.

Multiple matching: A bucket may appear in multiple join bucket pairs.

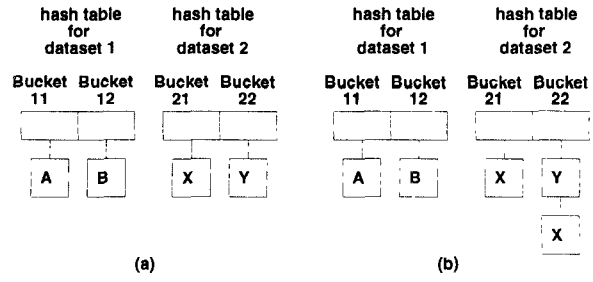

Figure 2: (a) A non-duplicative hash table produced by the multiple-matching approach. (b) A duplicative hash table produced by the multiple-assignment approach.

The benefit of the multiple-assignment approach is algorithmic simplicity. The only conceptual difference from relational hash-joins is that a partition function now maps an input object to a set of buckets instead of a single bucket. Its drawback is that data sizes may be inflated after partitioning, resulting in more I/O.

No size inflation occurs with multiple-matching. However, we may need to read a bucket more than once during the join phase, so we must schedule join-phase reads carefully so as to minimize disk I/O. Also, identifying join buckets pairs is no longer trivial.

For the example in Figure 1, the multiple-matching approach may hash the objects as in Figure 2(a). Since object X in dataset 2 overlaps both objects A and B in dataset 1, the join bucket pairs are $(11, 21), (12, 22)$ and $(12, 21)$. During the matching phase, we must be able to identify these bucket pairs.

There still remain scheduling and buffer-management problems. For example, suppose the buffer can hold only two objects and we join buckets pairs in the order $(12, 22), (11, 21)$ and $(12, 21)$. When bucket 12 is needed the second time, it would have been flushed to disk and must be read from disk again. We must either endure the buffer thrashing problem or carefully schedule the processing of join buckets pairs.

Using the same example, multiple-assignment may assign X into both buckets 21 and 22 (see Figure 2(b)). During the matching phase, we need only match bucket pairs $(11, 21)$ and $(21, 22)$. With this approach, buffer thrashing is not a problem. However, object X is written and read twice.

We believe duplication is particularly suitable in the context of spatial joins. It has also been used with spatial indices [20]. The use of duplication in spatial index trees complicates insertion and deletion of objects. However, this is not a drawback with data structures for joins, since they are constructed and used once. It is never necessary to update them once they are built. Hybrids of these two main approaches may also be appropriate for some situations.

4.2 Spatial Partition Function

Our spatial partition functions are defined by two components: a *assignment function* and a set of *bucket extents*, one bucket extent associated with each bucket. The assignment function maps a spatial object to a set of buckets. The maximum cardinality of this set is a design parameter. Quite unlike relational hash-joins, we allow the partition functions for the inner and outer dataset to differ. This feature is useful in addressing the coherent-assignment problem. A bucket extent corresponds to a region (not necessarily contiguous) in the space where the spatial join is to be computed, and is used in assigning objects to buckets.

Spatial objects can be hashed on various properties with different degrees of efficacy. For example, they may be hashed based on the bit patterns of their binary representations, or be transformed into points in higher dimensional spaces and hashed on coordinates in these spaces. Our implementation focuses on hashing spatial objects on their locations in the original space, though our framework is not limited to hashing in the original space. Since spatial objects are generally stored with coordinate values in their original space, our approach requires no additional transformations.

Bucket extents serve two main functions: (1) they are used by the assignment function to determine the buckets an input spatial object is mapped to, and (2) they are used by the join phase to identify join bucket pairs. As we hash in the original space, a bucket extent corresponds to a region in the space where spatial objects logically reside. A bucket extent is generally suggestive of the spatial coverage of the objects assigned to the bucket. However, it is not necessarily a bounding box for the objects assigned to its associated bucket. There is also no obvious counterpart to bucket extents in relational hashing.

The assignment function assigns a spatial object to buckets based on relationships between object attributes and bucket extents. For example, it may assign an object to a bucket whose extent overlaps it the most, or to all buckets whose extents overlap it.

The design of a spatial hash-join algorithm consists of determining: (1) the set of bucket extents, (2) the assignment function, and (3) the join bucket pairs. The choices made for these parameters can realize either the multiple-assignment or the multiple-matching approach, and affect efficiency.

4.2.1 Choosing Bucket Extents

The partition function must divide the input datasets into approximately equal-sized buckets by properly choosing the bucket extents and the assignment function. In addition to the locations, sizes and shapes of the individual bucket extents, the following parameters should be decided for the set of bucket extents:

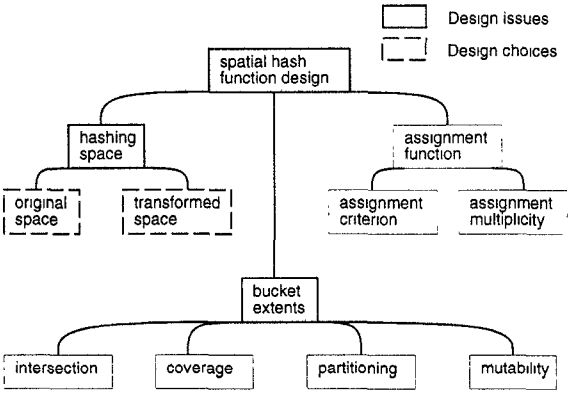

Figure 3: Design space for partition functions.

Intersection: do extents overlap?

Coverage: does the union of all extents cover the map area?

Partitioning: are extents determined by partitioning the space statically or based on object distribution?

Mutability: are the bucket extents updated as data are inserted into buckets? If so, how?

4.2.2 Choosing the Assignment Function

An assignment function maps an input object to a set of buckets based on its relationship with the bucket extents. An object may overlap or be contained by a bucket's extent, but still not be assigned to it. Assignment function design consists of two related aspects:

Assignment criterion: the spatial relationship between data objects and bucket extents upon which assignment of objects to buckets is based.

Assignment multiplicity: the number of buckets an input object can be assigned to.

Some examples of assignment criterion are to assign an object to: (1) all buckets whose extents overlap it, (2) to all buckets whose extents fully contain it, or (3) to the bucket whose center is nearest the object. The assignment criterion may comprise several rules, depending on the assignment multiplicity. When the single-assignment principle is used, there must be a series of tie-breaking assignment rules to choose a single bucket when several buckets satisfy the primary assignment rules. Figure 3 summarizes the design space discussed so far for the partition functions.

4.2.3 Identifying Join Bucket Pairs

In principle, an inner bucket must be matched with every outer bucket that may contain objects overlapping the inner bucket's objects. We may be able to reduce this matching space significantly in practice depending on the design of the inner and outer partition functions.

When the multiple-assignment principle is chosen and the single-matching property preserved, each inner bucket need only be joined with its corresponding outer bucket. When the multiple-matching principle is applied and each bucket may appear in many join bucket pairs, known properties of the bucket extents and assignment functions can be used to identify join buckets pairs. For example, if we know that every bucket's extent fully contains all objects assigned to it, each inner bucket need only be joined with the outer buckets whose extents overlap its extent. Methods for identifying join-bucket pairs thus depend on the partition function, and are considered components of its design.

4.3 Examples

We now present two straightforward examples for intersection join based on our framework. The first adopts the multiple-assignment approach, the second the multiple-matching approach.

4.3.1 Example 1

Bucket extents: Tessellate the map area with equal, non-overlapping regions, for example, with a regular grid of n cells. Each of these cells is the extent of one bucket. The inner and the outer datasets have the same set of bucket extents. The bucket extents are immutable.

Assignment function: A data object is assigned to all buckets whose extents overlap it. An object may thus be assigned to multiple buckets.

Join-bucket pairs: Join each pair of inner and outer buckets with the same extent (grid cell). This method may produce spurious matching object pairs when a pair of matching objects overlap the same two bucket extents. We must eliminate spurious results after the join.

The first drawback of this method is that bucket sizes may be very imbalanced. Depending on the spatial distribution of input objects, some buckets may have many objects, while others may be nearly empty. Second, we must duplicate objects when they fall on the boundary between two bucket extents, inflating data size. Finally, additional effort is necessary to remove redundant results.

The PBSM method [22] can also be interpreted using our spatial join framework. In some ways, its data partitioning scheme resembles the partition function of Example 1. PBSM divides the plane into a large number of grid cells, but instead of using a single grid cell as a bucket extent, it avoids imbalanced buckets by grouping several possibly non-contiguous cells to derive the equivalent of a bucket extent.

4.3.2 Example 2

Bucket extents: Start with n equal-sized, regular grid cells covering the whole map area, each cell being a bucket extent. The inner and outer datasets have the same set of *initial* bucket extents. As objects are assigned to a bucket, its extent is enlarged to enclose them. Thus, the final bucket extents for the inner and outer datasets may be different.

Assignment function: Each object is assigned to the bucket whose extent is enlarged the least after assignment, choosing the bucket whose extent has the nearest center of gravity if there are ties.

Join bucket pairs: Join each pair of inner and outer buckets whose extents overlap.

The drawback of this method is that during the join phase, we must track the bucket pairs to be joined. A bucket may participate in multiple join bucket pairs. If the buffer size is not large enough, a bucket may need to be read in several times during the join phase.

5 Our Design Choices

Several choices for the above design parameters are feasible for implementing a given spatial join. We do not seek an optimal method, since our goal here is only to demonstrate the feasibility of our approach and that it yields convincing performance gains. The optimal spatial hash-join method may be different for different join predicates and for input dataset with different characteristics, and will be the subject of future research.

We now design a spatial hash-join method for the intersection join using our framework. The resulting method is simple in design and efficient when implemented (see Section 6). We adopt the multiple-assignment approach because of its conceptual simplicity. It differs from the relational hash-join only in that its partition function maps a data object to multiple buckets. We expect that only a small amount of effort will be required to modify hash-join implementations in existing relational databases to realize this method.

We hash in the space where the objects reside, thus avoiding the costs of transforming representations. Our bucket extents are allowed to overlap, do not cover the whole object-distribution space, and based on object distribution in space.

This leaves open the initial values and the mutability of the bucket extents, and the assignment functions. Here we discuss only choices that affect the correctness of our methods. Design choices affecting only the efficiency are discussed in Section 6. Our choices for the inner dataset are:

Bucket extents: Initial value: see Section 6. Mutability: bucket extents are updated to enclose all assigned objects.

Assignment function: Assignment criterion: see Section 6. Multiplicity: Each object is assigned to exactly one bucket.

Under these choices a bucket extent is also a bounding box of the objects assigned to it. However, an object contained in the extent of a bucket may not belong to that bucket. The design choices for the outer dataset are:

Bucket extents: Initial value: Outer bucket extents = final inner bucket extents. If two inner buckets have the same extent, relabel the extents and associate one outer extent with each inner extent. Mutability: extents are immutable.

Assignment function: Assignment criterion: An object is assigned to all buckets whose extents overlap it. Multiplicity: An object may be assigned to multiple buckets.

Our outer assignment criterion serves to reduce the total number of outer dataset objects to be processed in the join phase, since an object not overlapping any bucket extent need not be assigned at all (see Section 6). Identifying the join bucket pairs is trivial given these choices of parameters. Each inner bucket simply pairs with the outer bucket with the same bucket extent.

We now show that our design produces correct and non-redundant results for intersection joins. When the the join phase begins, the corresponding buckets have the same extents. Let the inner buckets be $A_1, A_2, \ldots, A_k$, and the corresponding outer buckets $B_1, B_2, \ldots, B_k$.

Theorem 1: For each inner dataset object in A_i, all outer dataset objects that overlap it can be found in B_i.

Proof: We show that no outer dataset object outside B_i can overlap any object in A_i. If an outer dataset object p does not belong to B_i, it does not overlap B_i's extent. Since B_i and A_i have the same extent, p does not overlap A_i's extent. Since all inner dataset objects in A_i are contained by A_i's extent, p does not overlap any inner dataset objects in A_i. □

Theorem 2: Joining all bucket pairs identified as above produces the exact answer of the join.

Proof: We know that a bucket contains at most one copy of any object. Since each inner object is assigned to only one inner bucket, and an inner bucket is joined with exactly one outer bucket, an inner dataset object can meet any outer dataset object at most once when considered for join. □

Any spatial hash join method following the above design choices will produce the correct answer for intersection joins. The actual choices for initial shape and location of inner bucket extents and the assignment criterion for the inner assignment function may affect efficiency, but not algorithm correctness. These issues are discussed further in Section 6.

This design can also be modified very easily to implement *containment* queries. If we want to find object pairs such that the inner dataset object contains the outer dataset object, we modify the assignment criterion for the outer assignment function to assign objects to buckets whose bucket extents contain the object, and to look for containment instead of overlapping object pairs during the join phase. Every other aspect of the algorithm stays the same.

6 Implementation

Producing equal-sized buckets is crucial to the performance and stability of hash-join methods. For the above design choices, it is best to choose the initial inner bucket extents and an inner assignment criterion that lead to a set of final inner buckets with the following properties: (1) each bucket contains approximately an equal number of objects, (2) bucket extents overlap as little as possible, and (3) the total area of the bucket extents is minimized. The two latter requirements arise since the outer bucket extents are set to final inner bucket extents. When bucket extent overlaps are minimized, the probability of an outer dataset object being assigned to multiple buckets is minimized. When the total area of bucket extents is minimized, the probability of not assigning an outer dataset object to any bucket is maximized. These requirements are very similar to the requirements for nodes of spatial index trees. This suggests that our partition functions would benefit from techniques developed for constructing spatial index trees.

6.1 Determining Inner Bucket Extents

We choose rectangular bucket extents in our implementation, and use the *bootstrap seeding* technique [9] to identify their initial configuration. *Seeding* tailors a seeded tree for a join by setting its initial (or *seed)* levels (see Figure 4). Bootstrap-seeding builds a seeded tree directly from its underlying dataset. Readers are referred to [9] for full details of this technique.

All key seed-level information is contained in the *slots* [8, 9]. Using our present terminology, a slot can be considered to represent a bucket extent and a set of spatial objects. Initially, slot extents are points, and the set of objects is empty. As objects are assigned to the slots, their extents are updated to enclose the objects. The slot extents determine how objects are assigned to slots, and also largely determine the how effectively data are grouped into subtrees. Determining the number and initial extents of slots involves three steps:

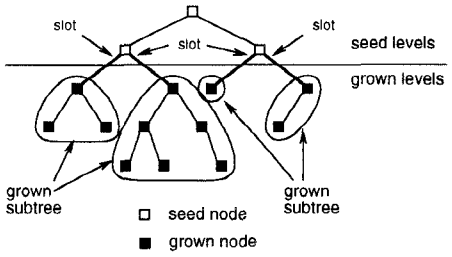

Figure 4: Example of a seeded tree.

1. Determining the number of slots S. A formula describing the upper and lower bounds for choosing the number of slots was derived in [9]. In this work, we use the average of the upper and lower bounds as the number of buckets.
2. Sampling the input data set, the sample size being some multiple of the number of slots.
3. Placing the S slots in the map area using information from the sample. We identify S clusters among the sample objects, and choose their centers to be the slot locations. In [9], we examined several heuristics to identify clusters efficiently. We use the *nearest-center* heuristic in this implementation.

6.2 Inner Dataset Partitioning

We use the number and the initial extents of the slots determined by the bootstrap-seeding process as the number and the initial extents of the inner buckets. The initial bucket extents are points. As objects are assigned to a bucket, its extent enlarges to become the MBR of its member objects. Our inner assignment criterion assigns an object to a bucket whose extent enlarges the least. Bucket extent updating and the assignment criterion are similar to insertion into seeded trees. These heuristics reduce the total area and overlap of seeded-tree nodes, and are likely to do the same for bucket extents.

6.3 Outer Dataset Partitioning and the Join Phase

The outer bucket extents are immutable, and set to the final extents of their corresponding inner buckets. To partition the outer dataset, we assign a copy of an outer dataset object to every bucket whose extent overlaps it. If an object overlaps no bucket extents, we can discard it since it is irrelevant to the join result. We call this technique *bucket-extent filtering*. It is analogous to the seed-level filtering technique in [8, 9].

The partitioning of both the inner and the outer datasets uses a technique called *batch writes*. As in the seeded tree algorithm, we write all buckets larger than a pre-defined threshold to disk when the bucket contents grow to fill whole buffer. This technique writes buckets to disk mostly in sequential I/O, and contributes greatly to the performance of our method.

After the outer dataset is partitioned, pairs of inner and outer buckets with the same extent are joined. The sizes of buckets partitioned using our scheme are in general smaller than the buffer size. Thus, two buckets can be joined without additional I/O overhead. Since the filter step is usually I/O intensive and our work focuses on reducing I/O cost, we do not look for an optimal method for the bucket-bucket join.

The algorithm we use to join a pair of buckets is similar to the "brute force join" in [8] or the "indexed nested loop join" in [22]. We first construct the inner bucket into a quadratic-split-cost R-tree [17], and then probe the objects of the outer bucket against it. This approach requires only one bucket (instead of two) to exist in the buffer, and reduces the chances of buffer overflow during the join phase. If the R-tree so constructed does overflow the buffer, we use LRU buffer management as outer objects are probed against the tree.

Our spatial hash-join method can be summarized as follows:

P1 Obtain initial bucket extents for the inner dataset using bootstrap seeding. Assign each inner object to one bucket based on the bucket extents and the assignment criteria. Update bucket extents after each assignment.

P2 Set the outer bucket extents to the final inner bucket extents. Assign a copy of each outer object to every outer bucket whose extent overlaps the object.

J1 Join corresponding bucket pairs to produce result.

7 Experiments

We now study the behavior of our spatial hash-join method, and compare its performance with other spatial join methods. We conducted experiments on three methods: spatial hash join (***HJ***), seeded tree join (***SJ***), and R-tree join.

The seeded-tree join method (***SJ***) constructs two seeded-tree indices just prior to performing tree matching for the join. The first tree is constructed using the bootstrap-seeding technique, and the second is constructed by copy-seeding from the first tree [8]. We study three variants of R-tree joins in our experiments. ***RJ*** constructs two R-trees dynamically and then performs tree matching. To offer the R-tree join a clear advantage, we also considered an ideal case when the input data is stored so that there is no buffer thrashing at all during tree construction (though there may be some during tree matching). This method is denoted as ***RJ(I)***, and is implemented by disregarding all random read costs during tree construction. ***RJ(M)*** is given two pre-computed R-tree indices, and simply performs tree matching. ***RJ(M)*** is always the most

Method	Description
HJ	Spatial Hash Join
SJ	Bootstrap-seeding method
RJ	R-tree join, no pre-existing indices (both indices built before join)
RJ(I)	***RJ***, input datasets stored so no buffer thrashing while building indices
RJ(M)	R-tree join, two pre-existing indices

Table 1: Competing methods

Experiments	Parameter	Section
Basic Tests	Size	Section 7.2
	Clustering	Section 7.2
Stability Tests	Degenerate	Section 7.3
	Real-life	Section 7.3
Buffer size	Buffer size	Section 7.4

Table 2: Experimental parameters

efficient among the R-tree join variations, but we emphasize that it depends on pre-computed indices, and cannot be applied to all cases in practice. The tree matching step in ***SJ***, ***RJ***, ***RJ(I)*** and ***RJ(M)***, implements all the optimization techniques described in [21]. Table 1 summarizes these competing methods.

Our experiments have focused on the filter step. For simplicity, we assume the sizes of disk blocks, memory pages, and R-tree and seeded-tree nodes all to be 8K bytes. Unless otherwise specified, we used a buffer of 512K bytes. The data files are assumed to contain entries consisting of a 16-byte bounding box and a 4-byte object identifier. Since we focus on the filter step, we assume the output to consist 8-byte object identifier pairs. Also, since the filter step is I/O-bound, we focus on I/O costs in our measurements. The ratio of the cost of accessing one disk block randomly to that of accessing one disk block sequentially is assumed to be 5, unless otherwise specified.

7.1 Experimental Data

The range and nature of data and other parameters used in our study is shown in Table 2. A series of basic tests established the performance of the ***HJ*** method under conditions expected to be frequently encountered, and a series of tests confirmed the stability and robustness of the method. The stability tests included real-life data, as well as data designed to induce degraded performance in the algorithm.

We studied input datasets of varying sizes and degrees of spatial clustering. The degree of clustering was controlled by a simple scheme. When generating a data set of $x \times y$ objects, we first generated x *cluster rectangles*, whose centers were randomly distributed in the map area. We then randomly distributed the centers of y data rectangles within each clustering rectangle. We could control the degree of clustering of the data set by controlling the total area of the clustering rectangles. We denote the cover quotient of the clustering rectangles (total area of the clustering rectangles divided by the map area) by CCQ. The smaller the value of CCQ, the more clustered the data set.

The length and the width of each clustering rectangle were chosen randomly and independently to lie between 0 and a predefined upper bound. This upper bound controlled the total area of the clustering rectangles. The size and shape of data rectangles were similarly chosen using a smaller upper bound. When clustering rectangles or data rectangles extended over the boundary of the map area, they were clipped to fit into the map area. When a data rectangle extended over the boundary of its clustering rectangle, it was not clipped. In the experiments, the number of data objects per cluster was set to be 200, and the number of clustering rectangles was set according to the total number of data objects. Without loss of generality, the map region under study was assumed to range from 0 to 1 along both X and Y axes.

7.1.1 Choice of Data Size and Clustering

We conducted two series of basic experiments. In the first series, we fixed the cardinality of the outer dataset at 100,000, resulting in an R-tree of 4 levels, and varied the cardinality of the inner dataset from 20,000 to 80,000. The upper bound on the side length of clustering rectangles was set to 0.04. The resulting CCQ quotient of the clustering rectangles in the outer dataset was 0.2, meaning that the centers of all the data objects in the outer dataset were restricted to 20% of the map area.

To study the effect of spatial clustering of data on joins, we fixed the cardinality of the inner and outer datasets at 40,000 and 100,000, respectively, and varied the degree of clustering of the data sets. We adjusted the upper bound on side length of the clustering rectangles so that the CCQ of the outer dataset was 0.2, 0.4, 0.6, 0.8 and 1.0, respectively. The upper bound on side length of the clustering rectangles of the inner dataset was the same as that of the outer dataset in each experiment.

7.2 Basic Experiments

Figure 5 shows the results of various joins methods on an example join. As clearly shown in the figure, ***RJ*** is the worst method by far. This behavior occurs primarily because R-trees are not designed to be constructed all at once, and cause severe buffer thrashing when so constructed. This trend held in all our experiments, so

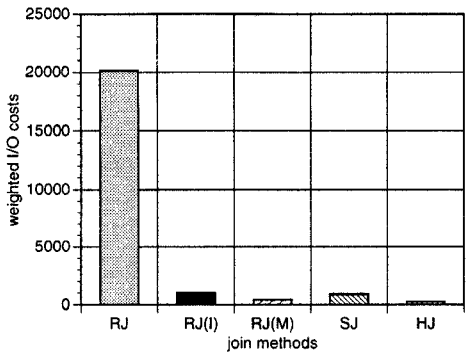

Figure 5: Join method costs. Datasets sizes are 100K (2M-bytes) and 40K objects (800K-bytes), respectively. $CCQ = 0.2$.

we drop ***RJ*** from further consideration as a competitor for ***HJ***.

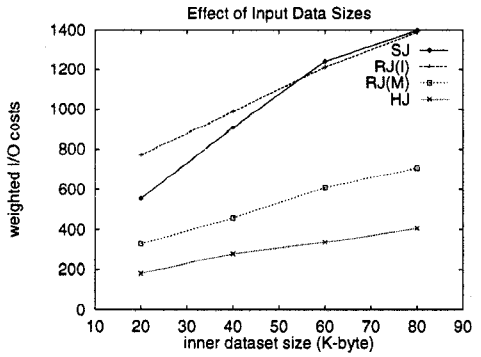

Figure 6: Join costs under different input dataset sizes.

Figure 6 shows the results of the first series of basic experiments, which varied input dataset sizes. ***SJ*** and ***RJ(I)*** incurred the highest costs. Although ***RJ(I)*** assumed no buffer thrashing during tree construction, the nature of its tree construction algorithm still caused its tree nodes to be written to disk using random I/O, increasing costs. It is noteworthy that while ***SJ*** did not make the idealized assumptions of ***RJ(I)***, its tree-construction techniques enable it to run faster than ***RJ(I)*** in many cases. ***RJ(M)*** ran faster than ***RJ(I)*** and ***SJ*** as it ignores tree-construction costs altogether.

The costs of ***HJ*** are much lower than even those of ***RJ(M)***, although ***HJ*** includes the costs of both partitioning the input datasets and joining them, while ***RJ(M)*** includes tree-matching costs only. The main reason is that the tree matching process of ***RJ(M)*** incurs random I/O, while the join phase of ***HJ*** incurs mainly sequential I/O. Even though ***RJ(M)*** accesses less data (only the pre-computed indices), the effects of random I/O make it more expensive. Thus, ***HJ*** is the method of choice even if the input datasets have pre-computed R-trees.

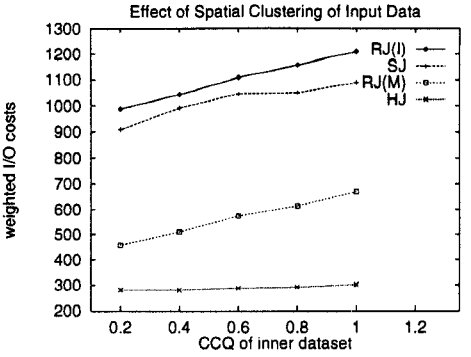

Figure 7: Effect of data clustering on join methods.

The second series of basic experiments studied the effects of spatial clustering on join costs (see Figure 7). Again, ***HJ*** demonstrates substantial performance gains over all other methods.

For all methods except ***HJ***, the processing costs were higher when data were less clustered spatially. This is because the degree of spatial clustering influences the number of tree nodes accessed during the tree matching step of these methods. ***HJ***, on the other hand, incurred almost constant costs as the degree of spatial clustering varied. This confirms that the costs of our spatial hash join method are easier to estimate than those of competing methods, facilitating query planning and optimization.

7.3 Stability Tests and Tests with Real-Life Data

Our next series of experiments compared the stability of various methods. These experiments were performed with datasets with cardinalities of 100,000 and 40,000.

In experiment "CLU-UNI", the inner dataset was a clustered dataset with $CCQ = 0.2$, and the outer dataset was a uniform dataset for which $CCQ = 1$. The experiment "UNI-CLU" worked the other way around. Experiment "EXCL" tested two spatially clustered but negatively correlated spatial datasets, both having $CCQ = 0.2$. Because of negative correlation, "EXCL" produced only 40K matched object pairs, instead of the approximately 70K pairs in other experiments.

We also ran tests with real-life data. These experiments joined two datasets extracted from the TIGER line files of the US Census Bureau [24]. The first dataset was a street map with 131,461 objects, the second a map of rivers and railway tracks with 128,971 objects. The MBRs of the objects were used. Experiment "REAL12" ran with the first dataset as the inner dataset and the second dataset as the outer dataset, while "REAL21" worked the other way around.

Figures 8 shows the results of these experiments. ***HJ*** ran as least 2.5 times faster than ***SJ***, and at least 3 times faster than ***RJ(I)***. Though comparisons with ***RJ(M)***

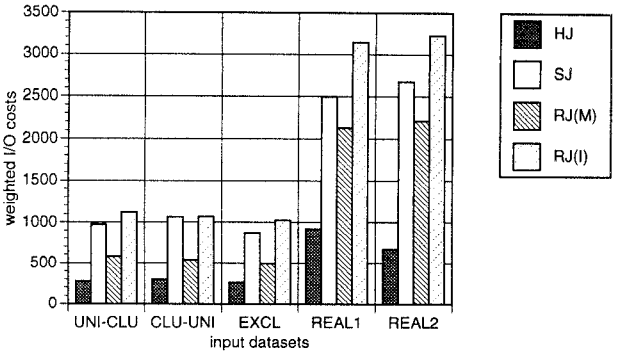

Figure 8: Stability tests and tests on real-life data.

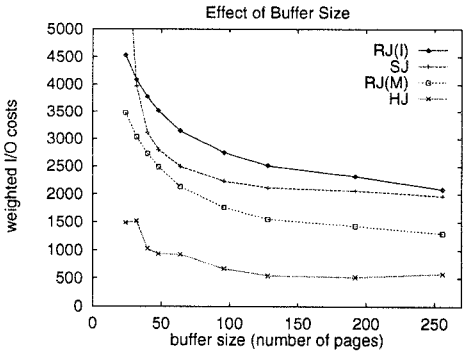

Figure 9: Buffer size effects (real-life data).

are unfair since it assumes pre-computed indices, ***HJ*** still runs approximately twice as fast as ***RJ(M)***.

Experiments "UNI-CLU", "CLU-UNI" and "EXCL" all used input data of the same size, but with different degrees of spatial clustering. Figure 8 again shows that while the costs for rival methods varied under different data characteristics, the costs for ***HJ*** were almost constant for these three sets of input data. This confirms that ***HJ*** is the most stable method among all tested.

7.4 Effects of Buffer Sizes and ratio r

We also studied the effects of buffer size on the join methods by varying buffer sizes from 24 pages (192K Bytes) to 256 pages (2M bytes). Figure 9 shows the results of experiments with the real-life datasets. Experiments with synthetic datasets show similar trends. In general, ***HJ*** performs the best, and ***RJ(M)*** comes in second, with ***SJ*** and ***RJ(I)*** being the worst. As expected, the join costs rose for all methods as the buffer size dropped. The costs for ***HJ*** stayed low and relatively stable through out the whole range of buffer sizes.

We also tested the effects of r, the ratio of random to sequential block access costs, on the join costs. We had set $r = 5$ in all our earlier experiments. Here we show the join costs for r in the range 1–30. Figure

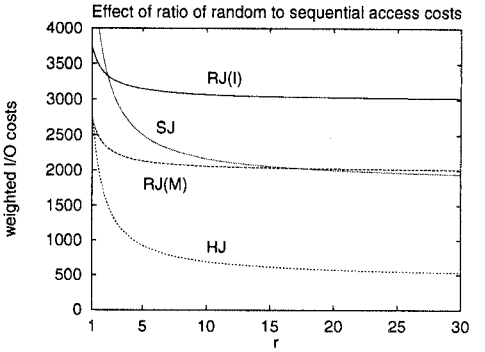

Figure 10: Effects of ratio r (real-life data).

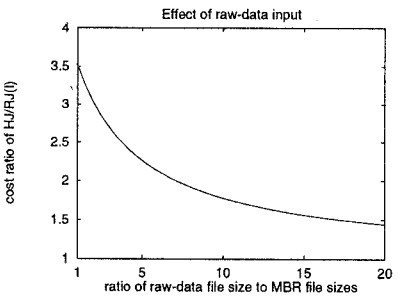

Figure 11: Projected performance given raw data.

10 shows the results of experiments with real-life data. Tests with synthetic data show very similar trends. ***HJ*** and ***SJ*** exploit the differences between random and sequential block access costs, so their performance improved faster than ***RJ(I)*** and ***RJ(M)*** as r increased. It is noteworthy that ***HJ*** began to outperform all other methods at very small values of r. This happened at $r \approx 2.08$ for the synthetic datasets, and at $r \approx 1.05$ for the real-life datasets. The costs for ***HJ*** are considerably lower than all other methods even for r as small as 5. As r increases to 30, the performance gaps between ***HJ*** and other methods widen continuously but not drastically. ***HJ*** does not require very large value of r to outperform other methods.

8 Discussion

The outer dataset size may be inflated since multiple assignment is used in the outer partition function. On the other hand, bucket filtering deflates the outer dataset size by discarding irrelevant objects during partitioning. In the 14 experiments that we ran using a 512K byte buffer, the number of additional copies produced was 17.5% of the original number of outer dataset objects, on average. Bucket-filtering eliminated an average of 12.7% of objects from the original outer dataset. The average net inflation of the outer dataset size was 5.2%. The bucket-filtering effect tends to be stronger when the outer dataset is much larger than

the inner dataset, and with spatially clustered data. Filtering actually reduced net outer dataset size by over 30% in some cases. The largest inflation from multiple insertion was 62.6%, and the largest net inflation was 61.6%. We observed that inflation and bucket-filtering effects can depend significantly on the number of and initial settings for inner bucket extents. More research is required on these issues.

An input dataset may comprise files containing raw data, MBR and object pointer pairs, and in some cases, pre-computed spatial indices. The MBR and index files tend to be similar in size, but raw data files may be many times larger. Our experiments assume input to be MBR files. If only raw data exists, we would need to compute MBRs on the fly. We expect this process to be I/O bound, and to increase read costs for all methods except ***RJ(M)***, which cannot run in this case because of its reliance on pre-computed R-trees. Although the cost increase is the same for all methods, it diminishes the the ratio by which ***HJ*** outperforms other methods. Figure 11 projects the performance gain of ***HJ*** over ***RJ(I)*** when input datasets exist only in raw data format. We assume that raw data files of all sizes generate the same MBR file, so that join costs are affected only by the costs of reading input. As the figure shows, even when the raw data is as much as 20 times larger than the MBR data size, ***HJ*** still runs substantially faster than ***RJ(I)***.

When pre-computed R-trees exist, ***RJ(M)*** comes closest to ***HJ***. If MBR files also exist, ***HJ*** outperforms ***RJ(M)*** substantially, as demonstrated by our experiments. When R-trees exist, but not MBR files, ***HJ*** can simply read the R-tree file, discard all internal tree nodes, and treat the leaf nodes as its MBR file. The performance advantages of ***HJ*** over ***RJ(M)*** will still hold.

HJ outperforms ***SJ*** and ***RJ(M)*** by a large factor in all cases, even though our comparison of ***HJ*** and ***RJ(M)*** is manifestly unfair since ***RJ(M)*** assumes pre-computed indices but ***HJ*** does not. Our experimental results clearly show that ***HJ*** is a very efficient join method whether or not pre-computed indices exist.

Patel and DeWitt [22] describe the implementation of the filter and refinement steps for their PBSM join method on the Paradise database system [25], and report both I/O and CPU times. Our focus has been on the filter step. The data partitioning phase in PBSM corresponds to the partition phase in our spatial hash-join framework (see Section 4.3.1). We can use the context of our framework to compare the respective strategies of PBSM and of our design. The PBSM equivalent of bucket extents are non-overlapping, immutable, cover the the whole map area, and are based on static space partitioning. Identical bucket extents and assignment functions are used for both input datasets. PBSM balances buckets by using extents that may comprise many non-contiguous subregions. It applies multiple assignment to both inner and outer datasets, so they may both be inflated after partitioning. Elimination of duplicate results is also necessary. In contrast, our method uses bucket extents that overlap and do not cover the whole map area. We balance bucket sizes by choosing bucket extents that are adaptive and based on the spatial distribution of objects. Our bucket extents are initialized by sampling input data, and evolve as input objects are assigned. Our partition functions for the inner and outer datasets are different. This asymmetry offers several benefits. First, multiple assignment need only be used for one dataset. The inner dataset is not inflated. Second, our design allows bucket filtering, which can reduce outer dataset size significantly. Its effects also serve to counteract data inflation, and even outweigh it in some cases. Third, our method produces no duplicate results, and requires no post-processing effort for their removal.

9 Conclusions

Although hash-join methods are efficient for implementing relational joins, it has been difficult to improve the efficiency of spatial joins using relational hash-join ideas due to the complexity of spatial joins predicates.

This paper overcomes the difficulties of applying hash-join paradigm to spatial join and defines a new framework for spatial hash-joins. Our spatial partition functions have two components: a set of bucket extents and an assignment function. An assignment function may assign a data object into multiple buckets. Furthermore, the partition functions for the two datasets may be different.

We have designed and tested a spatial hash-join method based on this framework. The partition function for the inner dataset is initialized by sampling the dataset, and evolves as data are inserted. The partition function for the outer dataset is immutable, and can assign objects into multiple buckets. The method mirrors relational hash-joins in other aspects. Our experiments show that our spatial hash-join method far outperforms earlier spatial join algorithms based on tree matching.

Acknowledgment:
The authors would like to thank Dr. T. Brinkhoff, Prof. H.-P. Kriegel and Prof. B. Seeger for generously providing the datasets used in our experiment with real-life data.

References

[1] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *New Generation Computing*, vol. 1, no. 1, pp. 66–74, 1983.

[2] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, "Implementation

techniques for main memory database systems," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 1–8, 1984.

[3] D. J. DeWitt and R. Gerber, "Multiprocessor hash-based join algorithms," in *Proceedings of VLDB 85*, pp. 151–164, Stockholm, 1985.

[4] M. Nakayama, M. Kitsuregawa, and M. Takagi, "Hash-partitioned join method using dynamic destaging strategy," in *Proceedings of the 14th VLDB Conference*, pp. 468–478, 1988.

[5] M. Kitsuregawa, M. Nakayama, and M. Takagi, "The effect of bucket size tuning in the dynamic hybrid grace hash join method," in *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pp. 257–266, Amsterdam, 1989.

[6] L. D. Shapiro, "Join processing in database systems with large main memories," *ACM Transactions on Database Systems*, vol. 11, no. 3, pp. 239–264, September 1986.

[7] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Computing Surveys*, vol. 24, no. 1, pp. 64–113, March 1992.

[8] M.-L. Lo and C. V. Ravishankar, "Spatial joins using seeded trees," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 209–220, Minneapolis, MN, May 1994.

[9] M.-L. Lo and C. V. Ravishankar, "Generating seeded trees from data sets," in *The Fourth International Symposium on Large Spatial Databases (Advances in Spatial Databases: SSD '95)*, Portland, Maine, August 26-29 1995, Springer-Verlag.

[10] J. Orenstein, "A comparison of spatial query processing techniques for native and parameter spaces," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 343–352, 1990.

[11] D. Rotem, "Spatial join indices," in *Proceedings of International Conference on Data Engineering*, pp. 500–509, Kobe, Japan 1991.

[12] W. Lu and J. Han, "Distance-associated join indices for spatial range search," in *Proceedings of International Conference on Data Engineering*, pp. 284–292, 1992.

[13] J. A. Orenstein, "Redundancy in spatial databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Portland, OR, 1989.

[14] J. Orenstein, "An algorithm for computing the overlay of k-dimensional spaces," in *Advances in Spatial Databases (SSD '91)*, O. Gunther and H.-J. Schek, editors, pp. 381–400, Zurich, Switzerland, August 28-30 1991, Springer-Verlag.

[15] O. Gunther, "Efficient computation of spatial joins," *Proceedings of International Conference on Data Engineering*, pp. 50–59, 1993.

[16] R. H. Guting and W. Schilling, "A practical divide-and-conquer algorithm for the rectangle intersection problem," *Information Sciences*, vol. 42, no. 2, pp. 95–112, July 1987.

[17] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 47–57, Aug. 1984.

[18] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 322–332, May 1990.

[19] C. Faloutsos, T. Sellis, and N. Roussopoulos, "Analysis of object oriented spatial access methods," *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 427–439, 1987.

[20] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-tree: A dynamic index for multi-dimensional objects," in *Proceedings of Very Large Data Bases*, pp. 3–11, Brighton, England, 1987.

[21] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using R-trees," *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 237–246, May 1993.

[22] J. M.Patel and D. DeWitt, "Partition based spatial-merge join," in *Proceedings of the 1996 ACM-SIGMOD conference*, Montreal, Canada, 3-6 June 1996.

[23] C. Faloutsos and Y. Rong, "Dot: A spatial access method using fractals," in *Proceedings of International Conference on Data Engineering*, pp. 152–159, 1991.

[24] B. of Census, "Tiger/lines precensus files: 1990 technical documentation," Technical report, Bureau of Census, Washington, DC, 1989.

[25] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J. Yu, "Client-server paradise," in *Proceedings of the 20th VLDB Conference*, Santiage, Chile, September 1994.