Paper Number 204

Size Separation Spatial Join

Nick Koudas *Kenneth C. Sevcik*
Computer Systems Research Institute
University of Toronto

Abstract

We introduce a new algorithm to compute the spatial join of two or more spatial data sets, when indexes are not available on them. Size Separation Spatial Join (S^3J) imposes a hierarchical decomposition of the data space and, in contrast with previous approaches, requires no replication of entities from the input data sets. Thus its execution time depends only on the sizes of the joined data sets.

We describe S^3J and present an analytical evaluation of its I/O and processor requirements comparing them with those of previously proposed algorithms for the same problem. We show that S^3J has relatively simple cost estimation formulas that can be exploited by a query optimizer. S^3J can be efficiently implemented using software already present in many relational systems. In addition, we introduce *Dynamic Spatial Bitmaps* (DSB), a new technique that enables S^3J to dynamically or statically exploit bitmap query processing techniques.

Finally, we present experimental results for a prototype implementation of S^3J involving real and synthetic data sets for a variety of data distributions. Our experimental results are consistent with our analytical observations and demonstrate the performance benefits of S^3J over alternative approaches that have been proposed recently.

Keywords: Spatial Join, Sort Merge Joins, Multiple-Key Indexes, Data Structures

1 Introduction

Research and development in Database Management Systems (DBMS) in recent decades has led to the existence of many products and prototypes capable of managing relational data efficiently. Recently there is interest in enhancing the functionality of relational data base systems with Object-Relational capabilities [SM96]. This means, among other things, that Object-Relational systems should be able to manage and answer queries on different data types, such as spatial and multimedia data. Spatial data are commonly found in applications like cartography, CAD/CAM and Earth Observation/Information systems. Multimedia data include video, images and sound.

In this paper we introduce a new algorithm to perform the *Spatial Join* (SJ) of two or more spatial data sets. Spatial Joins generalize traditional relational joins to apply to multidimensional data. In a SJ, one applies a predicate to pairs of entities from the underlying spatial data sets and performs meaningful correlations between them. Our algorithm, named *Size Separation Spatial Join* (S^3J), is a generalization of the relational *Sort Merge Join* algorithm. S^3J is designed so that no replication of the spatial entities is necessary, whereas previous approaches have required replication. The algorithm does not rely on statistical information from the data sets involved to efficiently perform the join and for a variety of data distributions offers a guaranteed worst case performance independent of the spatial statistics of the data sets. We introduce and describe the algorithm, analyzing its I/O behavior, and compare it with the behavior of previous approaches. Using a combination of analysis and experimentation with an implementation, we demonstrate the performance benefits of the new algorithm.

The remainder of the paper is organized as follows. Section 2 reviews relevant work in spatial joins and describes two previously proposed algorithms for computing spatial joins of data sets without indices. Section 3 introduces and describes Size Separation Spatial Joins. Section 4 presents an analysis of the I/O and processor requirements of the three algorithms and compares their performance analytically. In section 5, we describe prototype implementations of the three algorithms and present experimental results involving actual and synthetic data sets. Section 6 concludes the paper and discusses directions for future work.

2 Overview of Spatial Joins

We consider spatial data sets that are composed of representations of points, lines, and regions. Given two data sets, A and B, a spatial join between them, $A\ sp_\theta\ B$, applies predicate θ to pairs of elements from A and B. Predicates might include, *overlap*, *distance within* ϵ, etc. As an example of a spatial join, consider one data set describing parking lots and another describing movie theaters of a city. Using the predicate '*next to*', a spatial join between these data sets will provide an answer to the query: "find all movie theaters that are adjacent to a parking lot".

The shapes of spatial objects are rarely regular. In order to facilitate indexing and query processing, spatial objects are usually described by their Minimum Bounding Rectangle (MBR) or some other approximation [BKSS94]. As suggested by Orenstein [Ore86], spatial joins can be executed in two steps. In the first step, called the *Filter Step*, the predicate is evaluated on the spatial approximations of objects, and a list of candidate join pairs is produced. In the *Refinement Step*, the actual spatial objects corresponding to the candidate pairs are checked under the predicate.

There exists an extensive body of work on spatial join algorithms. For Grid Files [NHS84], an algorithm for doing spatial joins was developed by Rotem [Rot93]. Brinkhoff, et al. [BKS93] proposed an algorithm to perform the spatial join of two spatial data sets indexed with R-trees [Gut84][SRF87].

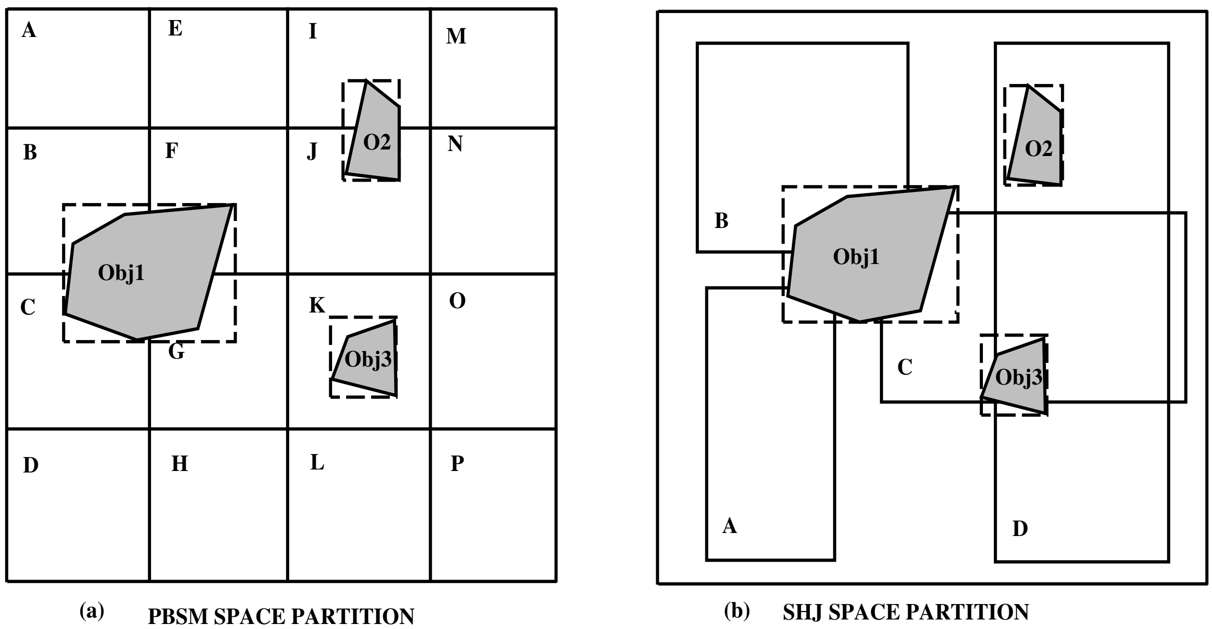


Figure 1: Space partition by the (a) PBSM and (b) SHJ algorithms

Sevcik and Koudas recently introduced an access method called *Filter Trees* and provided an algorithm to perform the Spatial Join of two data sets indexed with Filter Trees [SK96].

Lo and Ravishankar [LR94] studied the problem of computing spatial joins when an index exists only for one of the two data sets. They proposed a method called *Seeded Trees*, which makes use of an R-tree index already available on one data set to construct an index for the second data set dynamically. In subsequent work, they studied the problem of creating a seeded tree using sampling techniques [LR95].

Gunther [Gun93] presented an analytical model for evaluating the efficiency of spatial join computation, comparing nested loops and join indices. There are many efficient algorithms for computing the spatial join of two unindexed spatial data sets in the case where both sets fit in main memory. One example, which derives from computational geometry, is the *Plane Sweep* [PS85].

Two new algorithms have been proposed recently to solve this problem for the case where the data sets do not fit in main memory. Patel and DeWitt [PD96] introduced *Partition Based Spatial Merge Join* (PBSM) to compute the spatial join of two data sets without the use of indices. Lo and Ravishankar [LR96] also presented an algorithm for the same problem called *Spatial Hash Joins*. In the next subsections, we describe these two algorithms in greater detail.

2.1 Partition Based Spatial Merge Joins

Partition Based Spatial Merge Join (PBSM) is a generalization of the sort merge join algorithm. Given two spatial data sets, A and B, the algorithm uses a formula to compute a number of partitions into

Given two spatial data sets, A and B, and the number of tiles,

- Compute the number of partitions
- For each data set:
  1. Scan the data set;
  2. For each entity, determine all the partitions to which the entity belongs and record the entity in each such partition.
- Join all pairs of corresponding partitions (repartitioning, if necessary).
- Sort the matching pairs and eliminate duplicates

Figure 2: The PBSM Algorithm

which to divide the data space. These partitions act as buckets in hash joins. Once they are filled with data, only corresponding partitions for the two data sets must be processed to locate all candidate joining pairs. However, since the entities in the two data sets are in general not uniformly distributed, the number of objects that fall in various partitions will vary. To improve the chances of achieving balanced partition sizes, the algorithm partitions the space into a larger number of tiles and maps the tiles to partitions, either round robin or using a hash function.

A spatial entity might intersect two or more partitions. The algorithm requires replication of the entity in all the partitions it intersects. Once the first spatial data set has been partitioned, the algorithm proceeds to partition the second data set, using the same number and placement of tiles and the same tile to partition mapping function. Depending on the predicate of the spatial join, it might be the case that, during the partitioning of the second data set, a spatial entity that does not overlap with any tile can be eliminated from further processing since it cannot possibly join with any entities from the first data set. We refer to this feature of PBSM as *filtering*. Figure 1a presents a tiled space with three objects. Assuming four partitions, one possible tile-to-partition mapping is (A, B, E, F) to the first partition, (C, D, G, H) to the second, (I, J, M, N) to the third and (K, L, O, P) to the fourth. Under this scheme object Obj_1 will be replicated in the first and second partitions.

Once the partitions are formed for both spatial data sets, the algorithm proceeds to perform the join on partition pairs (repartitioning, if needed, to make pairs of partitions fit in main memory) and writes the results to an output file. Corresponding partitions are loaded in main memory and a plane sweep technique is used to evaluate the predicate. Since partitions may include some replicated objects, the algorithm has to detect (via hash or sort) and remove duplicates before reporting the candidate joining pairs. The complete algorithm is summarized in figure 2.

When both spatial data sets involved in the join are base sets and not intermediate results, one can adaptively determine the number of tiles one should use in order to achieve good load balance. For intermediate results, however, the appropriate number of tiles to use is difficult to choose, since statistical information is not available and an adaptive technique cannot be applied. If an inappropriate number of tiles is used, the algorithm still works correctly; however, using too few tiles may result in high load imbalance resulting in a lot of repartitioning, while using too many may result in an excessive number of replicated objects. Note that replication takes place in both data sets. The amount of replication that takes place depends on the characteristics of the underlying data sets, the number of tiles, and the tile to partition mapping function.

2.2 Spatial Hash Joins

Lo and Ravishankar proposed Spatial Hash Joins (SHJ) in order to compute the spatial join of two (or more) unindexed spatial data sets. The algorithm starts by computing the number of partitions [1] into which the data space should be divided. The computation uses a formula proposed by the same authors in earlier work [LR95]. Once the number of partitions is computed, the first data set is sampled. The centers of the spatial objects obtained from sampling are used to initialize the partitions. Then the first data set is scanned and the spatial entities are assigned to partitions based on the *nearest center* heuristic [LR95]. Each spatial entity is placed in the partition for which the distance from its center to the center of the partition is minimum. Once an entity is inserted in a partition, the MBR of the partition is expanded to contain the entity if necessary. When the MBR of the partition is expanded, the position of its center is changed. At the end of this process, the partitions for the first data set are formed. Notice that no replication takes place in the first data set.

The algorithm proceeds by scanning the second data set and partitioning it using the same partitions as adjusted to accommodate the first data set. If an entity overlaps multiple partitions, it is recorded in all of them, so replication of spatial entities takes place at this point. Any entity that does not overlap with any partition can be eliminated from further processing. Consequently *filtering* can take place in this step of the algorithm. Figure 1b presents one possible coverage of the space by partitions after the partitioning of the first data set. In this case, object Obj_1 of the second data set will have to be replicated in partitions A, B and C and object Obj_3 in partitions C and D.

After the objects of the second data set have been associated with partitions, the algorithm proceeds to join pairs of corresponding partitions. It reads one partition into main memory, builds an R-tree index on it, and processes the second partition by probing the index with each entity. If memory space is exhausted during the R-tree building phase, LRU replacement is used as outer objects are probed

[1] The authors use the term *slot* [LR96], but in order to unify terminology and facilitate the presentation, we use the term *partitions* throughout this paper.

Given two spatial data sets A and B,

- Compute the number of partitions
- Sample data set A and initialize the partitions
- Scan data set A and populate partitions, adjusting partition boundaries
- Scan data set B and populate partitions for B using the partitions of A and replicating where necessary.
- Join all pairs of corresponding partitions

Figure 3: The SHJ Algorithm

against the tree. The complete algorithm is summarized in figure 3.

2.3 Summary

Both PBSM and SHJ divide the data space into partitions, either regularly (PBSM) or irregularly (SHJ) and proceed to join partition pairs. They both introduce replication of the entities in partitions in order to compute the join. Replication is needed to avoid missing joining pairs in the join phase when entities cross partition boundaries. When data distributions are such that little replication is introduced during the partition phase, the efficiency of the algorithms is not affected. However, for other data distributions, replication can be unacceptably high, and can lead to deterioration of performance. Prompted by the above observation, in this paper, we present an alternative algorithm that requires no replication. We experiment with data distributions that can lead to increased replication using the previously proposed algorithms and we show the benefits of avoiding replication in such cases.

3 Size Separation Spatial Join

Size Separation Spatial Join derives its properties from the Filter Tree join algorithm [SK96]. Filter Trees partition spatial data sets by size. Given two spatial data sets, their join can be computed with minimal effort, requiring that each page of the data sets be read only once. S^3J constructs a Filter Tree partition of the space on the fly without building complete Filter Tree indices. The level j filter is composed of $2^j - 1$ equally spaced lines in each dimension. The level of an entity is the highest one (smallest j) at which the MBR of the entity is intersected by any line of the filter. This assures that large entities are caught at high levels of the Filter Tree, while most small entities fall to lower levels.

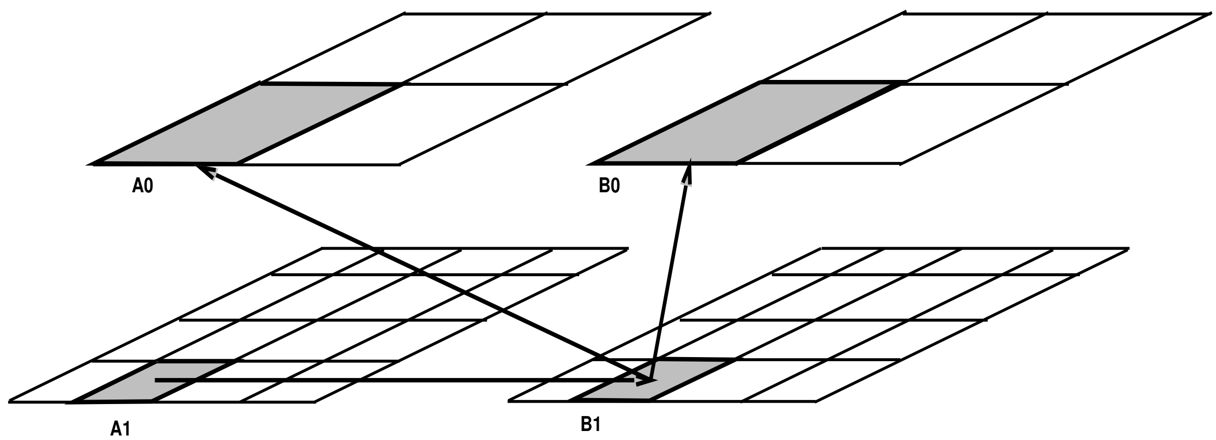


Figure 4: Space Partition by S^3J

- x_l, y_l
- x_h, y_h
- Hilbert(x_c, y_c)
- (pointer to) the actual entity

Figure 5: Format of entries in a level file

3.1 S^3J Algorithm

Denoting the opposite corners of the MBR of an entity by (x_l, y_l) and (x_h, y_h), S^3J uses two calculated values:

- Hilbert(x_c, y_c), the Hilbert value of the center of the MBR (where $x_c = \frac{x_l+x_h}{2}$, $y_c = \frac{y_l+y_h}{2}$) [Bia69].
- Level(x_l, y_l, x_h, y_h), the level of the Filter Tree at which the entity resides (which is the number of initial bits in which x_l and x_h as well as y_l and y_h agree) [SK96].

Given two spatial data sets, A and B, S^3J proceeds as follows. Each data set in turn is scanned and partitioned into *level files*. For each entity, its level, Level(x_l, y_l, x_h, y_h), is determined, and an entry of the form shown in figure 5 is composed and written to the corresponding level file for that data set.

The memory requirement of this phase is just $L+1$ pages where L is the number of level files (typically, 10 to 20) for the data set being partitioned. One page is used for reading the data set, and L are used for writing the level files.

Next, each level file for each data set is sorted so that the Hilbert values of the entries are monotonically nondecreasing. The final step of the algorithm is to join the two sets of sorted level files. The join is accomplished by performing a synchronized scan over the pages of all level files and reading each page once, as follows: Let $A^l(H_s, H_e)$ denote a page of the l-th level file of A containing entities with Hilbert values in the range (H_s, H_e). Then for level files $l = 0, \ldots, L$:

- process entries in $A^l(H_s, H_e)$ with those contained in $B^{l-i}(H_s, H_e)$ for $i = 0, \ldots, l$.
- process entries in $B^l(H_s, H_e)$ with those in $A^{l-i}(H_s, H_e)$ for $i = 1, \ldots, l$.

Figure 4 shows two levels of the space segmentation on which S^3J is based and presents the intuition behind the algorithm. S^3J divides the space in multiple resolutions as opposed to PBSM and SHJ which partition the object space at a single level. S^3J takes advantage of this space partitioning scheme and is able to perform the join while reading each page only once. Partitioning the space in multiple resolutions and placing each object at a level determined largely by its size, the algorithm can determine which pages are actually needed at each step. Figure 4, presents two data sets, A and B, each composed of two level files after being processed by S^3J. Partition A_1 from data set A needs to be processed against partitions B_1 and B_0 of data set B *only*. Similarly, partition B_1 of data set B has to be processed only with partition A_0 of A. No further processing for these partitions is necessary since no other overlapping pairs are possible.

Figure 6 summarizes the S^3J algorithm. Notice that no assumptions about the statistical properties of the data set are made. The algorithm can be applied either to base spatial data sets or to intermediate data sets without any modification. While we choose to use Hilbert curves to order level files, any curve that recursively subdivides the space will work (e.g., z-order, gray code curve, etc). Notice that the computation of the Hilbert value is not always necessary. The Hilbert values can be computed at the time entities are inserted and become a part of the descriptors of each spatial entity at the expense of storing them. For base spatial data sets this is probably a good choice. When the spatial data sets involved are derived from base sets via a transformation that changes the entity's physical position in the space or creates new entities, the Hilbert values can be recomputed.

The implementation of the S^3J algorithm is relatively straightforward. Partitioning the data sets involves only reading each entity descriptor and routing it to the appropriate level file (buffer page) based on examining the bit representations of the coordinates of the corners of its MBR. Sorting each level file, based on the Hilbert value of the center of the MBR of each entity, can be done with a sort utility commonly available in database systems. Finally, the synchronized scan of the level files strongly resembles an L-way merge sort (which can be implemented in a couple hundred lines of code).

3.2 Dynamic Spatial Bitmaps for Filtering

Both PBSM and SHJ are capable of *filtering*, which makes it possible to reduce the size of the input data sets during the partitioning phase. S^3J as described, performs no filtering since the partitioning

Given two spatial data sets A and B,

- Scan data sets A and B and for each entity:
  1. Compute the Hilbert value of the entity, $H(x, y)$.
  2. Determine the level at which the entity belongs and place its entity descriptor in the corresponding level file.
- For each level file,
  1. Sort by Hilbert value
- Perform a synchronized scan over the pages of level files.

Figure 6: Size Separation Spatial Join Algorithm

of the two data sets is independent. No information obtained during the partitioning of the first data set is used during the partitioning of the second.

S^3J can be extended to perform filtering by using *Dynamic Spatial Bitmaps* (DSB). DSB is similar to the technique of bitmap join indices in the relational domain [Val87] [OG95] [O'N96]. However, DSB is tailored to a spatial domain.

S^3J dynamically maps entities into a hierarchy of level files. Given a spatial entity, pages from all the level files of the joining data set have to be searched for joining pairs, but, as indicated in the previous section, this is done in a very efficient manner.

DSB constructs a bitmap representation of the entire data space as if the complete data set were present in one level file. A bitmap is a compressed indication of the contents of a data set. In the relational domain, using a bitmap of N bits to represent a relation of N tuples, we can perform a mapping between tuples and bits. Using this mapping we can obtain useful information during query processing. For example we could, by consulting the bitmap, check whether tuples with certain attributes exist. Now consider a two dimensional grid. In a similar manner, we can define a mapping between grid cells and bits of a bitmap. In this case the bitmap could, for example, record whether any entity intersects the grid cell or not.

To support filtering in S^3J, we use a bitmap corresponding to level l. At level file, l, there are 4^l partitions of the space, so the bitmap, M, will have 4^l one-bit entries. Initially all the bit entries of M are set to zero. Then, during the partitioning phase, for each spatial entity, e, that belongs to level file l_e and has Hilbert value $H_e^{l_e}$:

- If $l \leq l_e$, we transform the Hilbert value, $H_e^{l_e}$, of e into H_e^l (by setting to zero the $l - l_e$ least significant bits of $H_e^{l_e}$). We then set $M[H_e^l]$ to one.

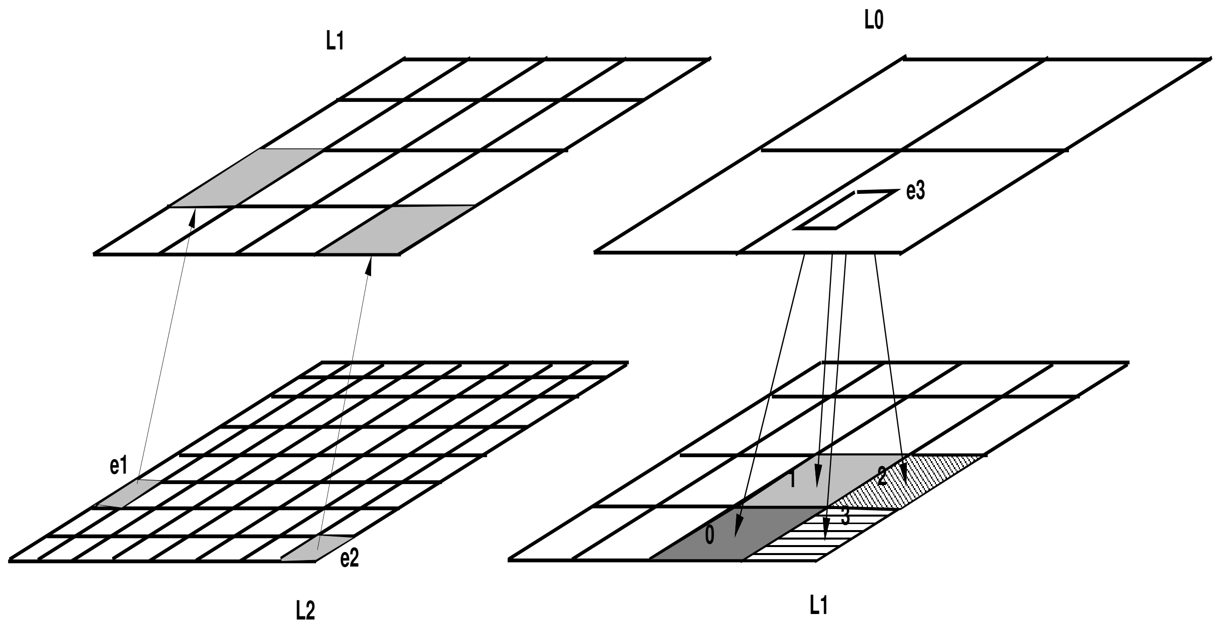


Figure 7: Example Operation of DSB

- If $l > l_e$ we have to compute the Hilbert values at level file l, $H^l_{e1}, H^l_{e2}, \ldots, H^l_{en}$, that completely cover e and set $M[H^l_{ei}], i = 1, \ldots, n$ to one. The computation of $H^l_{e1}, H^l_{e2}, \ldots, H^l_{en}$ can be performed either by determining all the partitions at level l that e overlaps and computing their Hilbert values, or by extending $H^{l_e}_e$ with all possible $l_e - l$ bit strings.

The operation described above essentially projects all entities onto level file l. Then, during the partitioning of the second data set B, for each spatial entity e, the same operation is performed, but this time:

- If $l \leq l_e$, e is placed into level file l_e only if $M[H^l_e]$ is set to one.
- If $l > l_e$, e is placed into level file l_e only if at least one of the bits $M[H^l_{e1}], M[H^l_{e2}], \ldots, M[H^l_{en}]$ is set to one.

Figure 7 illustrates the operation of Dynamic Spatial Bitmaps. Entities, e_1 and e_2, existing in level file L_2, are projected to the higher level L_1 which, for the purposes of this example, is the level chosen to represent the bitmap. The corresponding bit of the bitmap are set to one, indicating that entities exist in that portion of the space. Similarly entity, e_3 from level file L_0 is projected to L_1. For e_3, since it overlaps partitions 0 and 1 of L_1 only those bits should be set to one. We can either calculate the partitions involved for each entity and set the corresponding bits or set all the bits corresponding to the partition that contains e_3 in L_0 which is faster but less accurate.

Consider again the example in figure 4. A spatial entity belonging in partition B_1 of data set B needs to be stored in a level file for data set B only if a spatial entity of data set A exists in partitions A_1 or A_0. Information about whether any spatial entity of data set A exists in any partition of any level file is captured by the bitmap.

The size of the bitmap depends on which level file is chosen as the base onto which to project the data space. For level file l, the size of the bitmap is 4^l bits. With a page of size 2^p bits, 2^{2l-p} pages are needed to store the bitmap. Assuming a page size of 2^{12} bits (4KB), using level file seven for bitmap construction will yield a bitmap of four pages. Using level eight will yield a bitmap of sixteen pages and so on. There is a tradeoff between the size of the bitmap and its effectiveness. Using a lower level file (larger j) will yield a more precise bitmap. However, this will increase the number of pages needed to store the bitmap and the processor time to manipulate it. As long as a spatial entity belongs in a level lower than the level file used to represent the bitmap, the Hilbert value transformation is very fast, since it involves a simple truncation of a bit string. However for spatial entities belonging to level files higher than the bitmap level file, several ranges of Hilbert values have to be computed and this will increase the processor time required. Alternatively, one might choose to extend $H_e^{l_e}$ with all possible $l - l_e$ long bit strings. This will offer a fast Hilbert value transformation, since only a bit expansion is involved, but will decrease the precision of the bitmap.

Several options for advanced spatial query processing are opened by the use of DSB. For example, if bitmaps exist for both data sets, a bitmap intersection operation will identify the exact ranges of interest to the join. S^3J could take advantage of these ranges while performing a synchronized scan over the pages of the data sets.

4 Analysis of I/O behavior

In this section we present an analytical comparison of the I/O behavior of S^3J, PBSM and SHJ. Table 1 summarizes the symbols used and their meaning. For the purpose of this analytic comparison, we assume a spatial data set composed of entities with square MBRs of size $d \times d$ that are uniformly distributed over the unit square.

4.1 Analysis of the three algorithms

4.1.1 S^3J I/O analysis

The Size Separation Spatial Join algorithm proceeds by reading each data set once and partitioning essentially according to size, creating $L_A + L_B$ level files. The number of page reads and writes for data

Symbol	*Meaning*	*Symbol*	*Meaning*
S_f	Size of File f in pages	M	Memory Size in Pages
J	Size of join result in pages	r_f	replication factor for data set f
D	Divisions of space	L_f	Number of level files for data set f
H	Processor time to compute a Hilbert value	C	Size of candidate pair list before sort
E	Object descriptor entries per page	B	Size of bulk reads from disk

Table 1: *Symbols and their meanings*

sets A and B in the scan phase will be:

$$2\ S_A + 2\ S_B \tag{1}$$

The factor of two accounts for reading and writing each data set.

In the sort phase, S^3J sorts each level file. Assuming a uniform distribution of squares, level file i will contain a fraction of objects given by:

$$f_i = \begin{cases} d(2-d) & i = 0 \\ 2^i d(1 - \frac{3}{4}2^i d) & i = 1, \ldots, k(d) - 1 \\ (1 - \frac{1}{2}2^{k(d)}d)^2 & i = k(d) \end{cases} \tag{2}$$

where $k(d) = \lceil -\log_2 d \rceil$ is the lowest level to which any $d \times d$ object can fall (since d must be less than 2^{-k}) [SK96]. Then the expected size of each level file i for data set j will be about $S_{ij} = f_i S_j$, $i = 1 \ldots max(L_A, L_B)$, $j \in A, B$. Assuming that read requests take place in bulks of B pages from the disk, applying merge sort on the level file of size S_{ij} will yield a sort fan-in F of $\frac{M}{B}$ and $\lceil l_i = log_F S_{ij} \rceil$ merge sort levels (l_i will not commonly be one). The total number of page reads and writes of the sorting process is given by:

$$2 \sum_{i=1}^{L_A} l_A\ S_{iA} + 2 \sum_{i=1}^{L_B} l_B\ S_{iB} \tag{3}$$

Once the sorted level files are on disk, S^3J proceeds with the join phase by reading each page only once, computing and storing the join result, incurring:

$$S_A + S_B + J \tag{4}$$

page reads and writes The total number of page reads and writes of S^3J is the sum of the three terms above. The best case for S^3J occurs if each level file fits in main memory (i.e., $S_{ij} \leq M, \forall i$). In this case the total number of page reads and writes of the algorithm becomes:

$$5\ S_A + 5\ S_B + J \tag{5}$$

In its worst case, S^3J will find only one level file in each data set. In this case, the total number of page reads and writes will be:

$$3\ S_A + 3\ S_B + 2\ l_A\ S_A + 2 l_B\ S_B + J \tag{6}$$

Except for artificially constructed data sets, the largest of the level files would usually contain 10% to 30% of the entities in the data sets. If the Hilbert values are initially not part of the spatial entity's descriptor, then they have to be computed. This computation takes place while partitioning the data sets into levels. The processor time for this operation is:

$$H(S_A + S_B)E \tag{7}$$

Using a table driven routine for computing the Hilbert values, we were able to perform the computation in less than 10 μsec per value at maximum precision on a 133MHz processor, so $H \cong 10\mu$secs.

4.1.2 PBSM I/O analysis

The number of partitions suggested by Patel and DeWitt for the PBSM algorithm [PD96] is:

$$D = \frac{S_A + S_B}{M} \tag{8}$$

Defining the replication factor r_f as:

$$r_f = \frac{Data\ set\ size\ after\ replication\ and\ filtering}{original\ data\ set\ size\ (S_f)} \tag{9}$$

the number of page reads and writes during the partitioning phase is:

$$(1 + r_A)\ S_A + (1 + r_B)\ S_B \tag{10}$$

since the algorithm reads each data set and possibly introduces replication for entities crossing partition boundaries.

Entity replication will increase the data set size, making r_f greater than one, but filtering, will counteract that, reducing r_f, possibly to be even less than one for cases where the join is highly selective (i.e, where there are very few join pairs). Due to replication, the size of the output file that is written back to disk may be larger than the initial data set size. More precisely if A is the data set that is partitioned first, then $r_A \geq 1$ and $r_B \geq 0$. The amount of replication introduced depends on the data distributions of the data sets and the degree of dividing of the data space into tiles. Depending on data distributions, $1 \leq r_A \leq D$ and $0 \leq r_B \leq D$. Notice that r_B could be less than one depending on the partitioning imposed on the first data set. To illustrate the effects of replication, again assume uniformly distributed squares of size $d \times d$, normalized in the unit square. Then assuming a regular

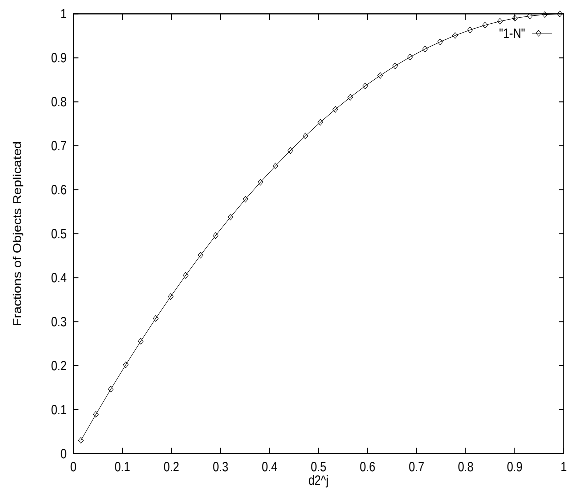


Figure 8: Fractions of Replicated Objects

partitioning of the unit square into sub squares of side 2^{-j}, the fraction, N, of objects falling inside tiles will be:

$$1 - d2^{j+1} + d^2 2^{2j} \tag{11}$$

assuming that $d \leq 2^{-j}$, so that the side of each square object is less than or equal to the side of each tile. As a result the fraction of objects replicated will be $d2^{j+1} - d^2 2^{2j}$. The amount of replication taking place depends on $d2^j$, since replication is introduced either by increasing the object size for constant number of tiles or by increasing the number of tiles for constant object size. Figure 8 shows the fraction of objects replicated as a function of $d2^j$. As $d2^j$ increases, the amount of replication that takes place increases.

The algorithm then checks whether corresponding partitions fit in main memory. Assuming that partitions have the same size and that each pair of partitions fits in main memory, the number of page reads and writes for this step is:

$$r_A\ S_A + r_B\ S_B + C \tag{12}$$

where C is the size of the initial candidate list. If partition pair i does not fit in main memory then it has to be repartitioned. Using equation (8) to compute the number of partitions we expect under a uniform distribution, half the partitions to require repartitioning. Using a hash function to map tiles to partitions, we expect the MBRs of partitions to be the same as the MBR of the original data file. Thus the fraction of replicated objects remains the same for subsequent repartitions. The total number of page IOs during the first partitioning phase is given by equation (10). Since on average half of the partitions will have to be repartitioned, the expected number of page IOs during the second partitioning phase will be:

$$\frac{(1+r_A)r_A S_A}{2} + \frac{(1+r_B)r_B S_B}{2} \tag{13}$$

For uniform data distributions, this is expected to offer acceptable size balance across partitions and pairs of corresponding partitions will fit in main memory. The algorithm proceeds to read all pairs of corresponding partitions and join them in main memory using plane sweep. The total number of page IOs for this phase will be:

$$\frac{(1+r_A)r_A S_A}{2} + \frac{(1+r_B)r_B S_B}{2} + C \tag{14}$$

where C is the size of the candidate list. After the join phase, the result of the join is stored on disk, but duplicate elimination must be performed since replication of entities may have occurred in both data sets. Duplicate elimination is achieved by sorting the join result. The number of page reads and writes during the sort is:

$$2\ J \sum_{i=0}^{l-1} \frac{C}{F^i\ M} = 2\ J\ \frac{C}{M}\ \frac{(1-\frac{1}{F^l})}{1-\frac{1}{F}} \tag{15}$$

where F is the fanout factor of the sort. The number of sort merge phases will be $l = \log_F C$. Since elimination of duplicates can take place in any phase of the sort we have to perform the summation over all sort merge phases, resulting in equation (15). If C fits in memory, the cost of page reads and writes during the sort (with duplicate elimination) will be $C + J$.

The total number of page reads and writes of the algorithm results if we sum all clauses above, taking into account whether intermediate results fit in main memory or not. The replication factors, r_A and r_B, play an important role in the total number of I/O's given above. Their value depends on the number of tiles in the space and the input data distributions.

4.1.3 Spatial Hash Joins

Assuming that data set A is to be processed with D partitions, the number of page reads and writes during sampling and partitioning of data set A is:

$$c\ D + 2\ S_A \tag{16}$$

where c is some integer and cD represents (an upper limit on) the random I/O performed while sampling set A. The number of page reads and writes during partitioning of data set B is:

$$(1+r_B)\ S_B \tag{17}$$

since all of data set B must be read and multiple r_B of its initial size must be written. After the partitioning phase, the algorithm joins the corresponding pairs of partitions. If the corresponding partitions for both data sets fit in main memory, both partitions will be read and then joined. The join can be done either using nested loops or by constructing an R-tree in main memory for the first partition and probing it with the elements of the second. If both partitions fit in main memory the number of page reads and writes during the join phase is:

$$S_A + r_B\ S_B + J \tag{18}$$

where the first two terms correspond to reads and the third to writes. However, with SHJ there is no guarantee that the partitions will be balanced in size or that they will fit in main memory. Moreover, the partition placement depends only on samples taken from one data set. A general analysis of SHJ is difficult, because its behavior depends on the distributions of the joined data. For uniformly distributed squares, an analysis similar to the one presented for PBSM can be applied. However, for specific data set sizes and main memory size, the number of partitions used by SHJ is much larger than the number used for PBSM. Consequently, the amount of replication required in SHJ is expected to be larger than that in PBSM. Assuming that partitions do not fit in main memory and that partitions are joined using nested loops, the number of page reads and writes during the join phase becomes:

$$\sum_{i=1}^{D}\left(\frac{S_{iA}}{M}\ S_{iB} + S_{iA}\right) \tag{19}$$

where S_{iA}, S_{iB} are the sizes of the partitions for A and B. Very little can be said about S_{iA} and S_{iB}. For uniformly distributed data sets, we expect $S_{iA} = \frac{S_A}{D}$ and $S_{iB} = r_B \times \frac{S_B}{D}$.

For SHJ, replication is introduced only for one of the two data sets involved. As in the case of PBSM, the value for the replication factor r_B plays an important role in the algorithm's performance. Notice that in the worst case $r_B = D$.

4.2 Analytical Comparison of the Algorithms

Using the formulas derived in the previous subsections, we perform an analytical comparison of the algorithms in terms of I/O behavior. The results are shown in figure 9. For this comparison, we assume that $S_A = S_B = 10000$ pages. For S^3J, an adversarial placement of objects based on knowing the position of the underlying grid could lead to a single level file with all objects in it. However if the adversary does not have knowledge of the exact position of grid lines (i.e., the grid can be shifted slightly), then even by placing objects in a non-independent fashion, extreme imbalance of level file size cannot be forced. Consequently, we take as a practical "worst case" a situation where 30% of the entities are in the largest of the level files. (This choice is supported by previous analysis [SK96].) To simplify the figures we assume that no duplicate join tuples are introduced during PBSM's operation, so the size of the intermediate result is equal to the size of the join result (that is, $C = J$). This favors PBSM, since fewer I/O operations are necessary, both when writing the intermediate result, and when sorting to eliminate duplicates.

Figure 9 presents the total number of page reads and writes predicted by the formulas, normalized to S^3J performance, for PBSM and SHJ as a function of the replication factors, r_A and r_B. This is a low selectivity join, since the size of the join result is twice the aggregate size of the two data sets. A word of caution is in order when interpreting figure 9: For two specific data sets, there is no reason to believe that the replication factor for the second data set, r_B , introduced by PBSM and SHJ is the same. As we pointed out earlier, we expect replication to be higher for SHJ. However, in order to present a comparison we represent r_B on a common axis for both algorithms.

S^3J does not introduce any replication so its performance is independent of replication factors. In addition, the buffering requirements of the algorithm during the partitioning and join phases are modest, being only a page per level file plus a page in the buffer pool for output tuples: $L_A + L_B + 1$. Increasing the available buffer size helps the sorting phase of S^3J (if the level files cannot fit in main memory).

In contrast, PBSM's performance depends heavily on the degree of replication as is evident in figure 9. As replication increases the performance of PBSM becomes worse. PBSM makes the partitions fit in main memory only when partitions are size balanced and the aggregate size of the data sets before partitioning is less than or equal to their aggregate size after partitioning. This is due to the fact that the number of partitions used for PBSM(equation (8)) does not take replication into account. It sets the number of partitions based on the size of the data sets before partitioning. When we increase the amount of memory available to the join, the part of PBSM's total number of page reads and writes that is affected, is the number of I/Os incurred while sorting. Since the sort can use a larger fanout, the number of I/Os is smaller.

SHJ introduces replication only in the second data set and, as a result, its I/O behavior is competitive to S^3J for higher replication factor values, as shown in figure 9. For the purposes of this analysis, each random disk access incurred via sampling during the partitioning phase of SHJ is assumed five times more expensive than the cost per page in a sequential access. SHJ typically uses a large number of partitions and thus frequently makes pairs of partitions fit in main memory during the join phase. Increasing the buffer size available by a factor f, causes an increase in the number of partitions by a factor which can be more than f of the number estimated before the increase. This will improve the join phase of the algorithm but will negatively affect its partitioning phase.

5 Experimental Comparison

In this section, we present experimental results from prototype implementations of all three algorithms. We include experimental results using combinations of real and synthetic data sets. We implemented all three algorithms on top of a common storage manager that provides efficient I/O. Several components common to all algorithms were shared between implementations, contributing to the fairness of the comparison of the algorithms at the implementation level. Specifically, the same sorting module is used by S^3J and PBSM, and all three algorithms use the same module for plane sweep.

All of our experiments were conducted on an IBM RS6000 model 43P (133MHz), running AIX with 64MB of main memory (varying the buffer size during experiments) with a Seagate Hawk 4 disk with capacity 1GB attached to it. The processor's SPEC ratings are SPECint95 4.72 and SPECfp95 3.76. Average disk access time (including latency) is 18.1 msec assuming random reads.

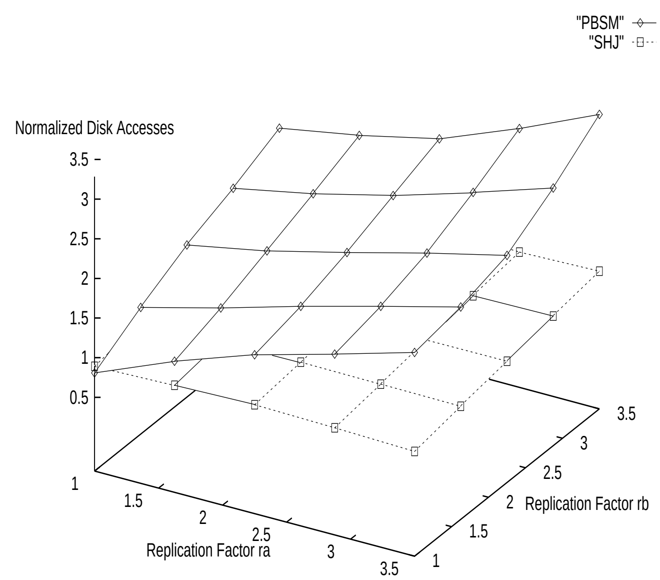


Figure 9: Analytical Comparison of Disk Accesses

We present and discuss sets of experiments, treating joins of synthetic and real data sets for low (many output tuples) and high (few output tuples) selectivity joins. For our treatment of S^3J, we assume that the Hilbert value is computed dynamically. If the Hilbert value were present in the entity descriptor initially, the response times for S^3J would be smaller than the ones presented by a small amount, reflecting savings of processor time to compute the values.

For PBSM, we demonstrate the effect of different parameters on the performance of the algorithm. We include results for various numbers of tiles. In all PBSM experiments, we compute the number of partitions using equation (8) as suggested by Patel et al. [PD96]. Similarly, SHJ performance depends on the statistical properties of the input data sets. We compute the number of partitions using the formula suggested by Lo and Ravishankar [LR95].

We present the times required for different phases of the algorithms. Table 2 summarizes the composition of the phases for the three algorithms. For the experiments that follow, unless stated otherwise, the total buffer space available is 10% of the total size of the spatial data sets being joined.

5.1 Description of Data Sets

Table 3 presents the data sets used for our experiments. All the data sets composed of uniformly distributed squares are normalized in the unit square. UN1, UN2 and UN3 have artificially low variability of the sizes of objects and consequently low coverage, 0.4, 0.9 and 1.6 respectively. Coverage is defined

Algorithm	*Phase*	*Contains*
S^3J	Partition	Reading, partitioning and writing the level files for both data sets
	Sort	Sorting (reading and writing) the sorted level files
	Join	Merging the sorted level files and writing the result on disk
PBSM	Partition	Reading, partitioning and writing partitions for both data sets
	Join	Joining corresponding partitions and writing the result on disk
	Sort	Sorting the join result with duplicate elimination and writing the result on disk
SHJ	Partition	Reading, partitioning and writing partitions for both data sets
	Join	Joining corresponding partitions and writing the result on disk
	Sort	none

Table 2: Phase Timings for the three algorithms

as the total area occupied by the entities over the area of the MBR of the data space. The LB and MG data sets contain road segments extracted from the TIGER/Line data set [Bur91]. The first (LB) presents road segments in Long Beach County, California. The second (MG) represents road segments from Montgomery County, Maryland and contains 39,000 line segments. Data set TR is used to model scenarios in which the spatial entities in the data sets are of various sizes. We produced a data set in which the sizes of the square spatial entities are generated according to a triangular shaped distribution. More precisely, the size of the square entities is, $d = 2^{-l}$ where l has a probability distribution with minimum value x_1 maximum value x_3, and the peak of the triangular distribution at x_2. As one would expect, the overlap among the entities of such a data set is high. TR contains 50,000 entities and was generated using $x_1 = 4, x_2 = 18, x_3 = 19$. CFD is a vertex data set from a Computational Fluid Dynamics model, in which a system of equations is used to model the air flows over and around aero-space vehicles. The data set describes a two dimensional cross section of a Boeing 737 wing with flaps out in landing configuration. The data space consists of a collection of points (nodes) that are dense in areas of great change in the solution of the CFD equations and sparse in areas of little change. The location of the points in the data set is highly skewed.

5.2 Experimental Results

5.2.1 No Filtering Case

We present and discuss a series of experiments involving low selectivity joins of synthetic and real data sets. Table 4 summarizes all the experimental results in this subsection and presents the response times of PBSM and SHJ normalized to the response time of S^3J as well as the replication factors observed

Name	*Type*	*Size*	*Coverage*
UN1	Uniformly-Distributed Squares	100,000	0.4
UN2	Uniformly-Distributed Squares	100,000	0.9
UN3	Uniformly-Distributed Squares	100,000	1.6
LB	Line Segments from Long Beach County, California	53,145	0.15
MG	Line Segments from Montgomery County, Maryland	39,000	0.12
TR	Squares of Various Sizes	50,000	13.96
CFD	Point Data (CDF)	208,688	-

Table 3: Real and Synthetic Data Sets used

Data Sets	*PBSM small #tiles*		*PBSM large #tiles*		*SHJ*	
used	*Response Time*	$r_A + r_B$	*Response Time*	$r_A + r_B$	*Response Time*	r_B
UN1,UN2	1.3	2.44	1.5	3.3	1.35	1.5
UN2,UN3	1.58	2.66	1.85	3.8	1.38	1.6
LB,LB'	1.9	2.4	2.34	3	1.33	1.62
MG,MG'	1.92	2.62	2.26	3.2	1.4	1.5
TR	2.32	4.92	3.1	7.8	2.65	10
CFD	1.75	4.2	1.96	4.6	3.04	4

Table 4: Join Response Times, normalized to S^3J Response Time and Replication Observed

for them.

The first two experiments involve data objects of a single size that are uniformly distributed over the unit square. For uniformly and independently distributed data, the coverage of the space is a realistic measure of the degree of overlap among the entities of a data set. From the first experiment to the second, we increase the coverage (using squares of larger size) of the synthetic data sets and present the measured performance of the three algorithms. For algorithms that partition the space and replicate entities across partitions, the probability of replication increases with coverage, for a fixed number of partitions.

Figure 10a presents the response time for the join oftwo uniformly distributed data sets, UN1 and UN2 containing 100,000 entities each. Results for PBSM are included for two different choices of tiling: the first choice is the number of tiles that achieves satisfactory load balance across partitions and the second is a number of tiles larger than the previous one.For S^3J the processor time needed to evaluate the Hilbert values accounts for 8% of the total response time. The partitioning phase is relatively fast, since it involves sequential reads and writes of both data sets while determining the output level of each spatial entity and computingits Hilbert value.

For PBSM, since we are dealing with uniformly distributed objects, a small number of tiles is enough to achieve balanced partitions. The greatest portion of time is spent partitioning the data sets. Most partition pairs do not fit in main memory and the algorithm has to read again and repartition those that they do not fit in main memory. Approximately half of PBSM's response time is spent partitioning the input data sets and the rest is spent joining the data sets and sorting (with duplicate elimination) the final output.

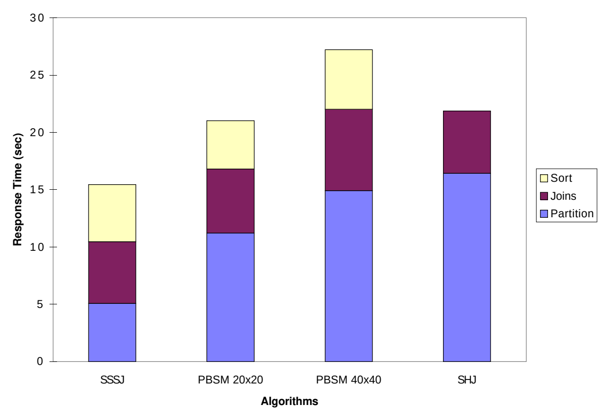
SHJ uses more partitions than PBSM does for this experiment. The large number of partitions covers the entire space and introduces overlap between partition boundaries. The algorithm spends most of its time sampling and partitioning both data sets. As is evident from figure 10a, the partitioning phase of SHJ is more expensive than the corresponding phase of S^3J and a little more expensive than that PBSM with large tiles. The join phase, however, is fast since all pairs of partitions fit in main memory and due to less replication, fewer entities have to be tested for intersection.

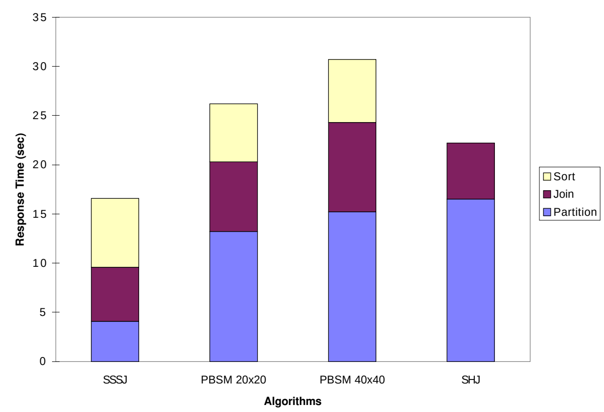
Figure 10b presents the results for the join of UN2 and UN3. The impact of higher coverage in UN3 relative to UN1 affects S^3J only in processor time during the join phase. The portion of time spent partitioning into levels and sorting the level files is the same. Although the partitioning times remain about the same, join time and sorting time increase according to the data set sizes. For SHJ the larger replication factor observed increases I/O as well as processor time in the partitioning and join phases. Due to the increased replication, the join phase of SHJ is more costly than in the previous experiment.

Figures 11a and 11b present results for joins of data sets LB and MG. For each of LB and MG, we produce a shifted version of the data set, LB' and MG', as follows: the center of each spatial entity in the original data set is taken as the position of the lower left corner of an entity of the same size in the new data set.

Figure 11a presents performance results for the join of LB and LB'. For S^3J, the time to partition and join is a little more than the time to sort the level files. When decomposed by S^3J, LB yields 19 levels files. The largest portion of the execution time is spent joining partition pairs. PBSM's performance is worse with more tiles due to increased replication. In this case, the join result is larger than both input data sets, so PBSM incurs a larger number of I/Os from writing the intermediate result on disk and sorting it. Not all partitions fit in main memory (because of the non-uniformity of the data set) and SHJ has to read pages from disk during the join phase. Figure 11b presents the corresponding experiment involving the MG and MG' data sets. Similar observations hold in this case.

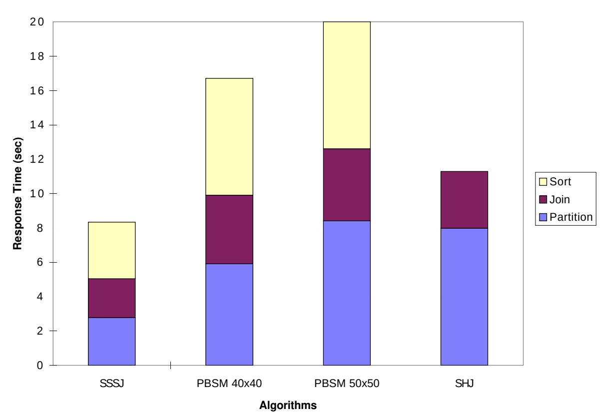
The experiments described above offer intuition about the trends and tradeoffs involved with real and synthetic data sets with moderate and low coverage. With the following experiment, we explore

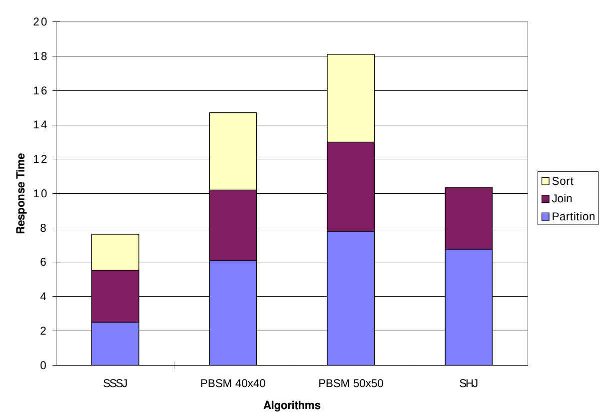




(a) UN1 join UN2 (coverage 0.4 and 0.9) (b) UN2 join UN3 (coverage 0.9 and 1.6).

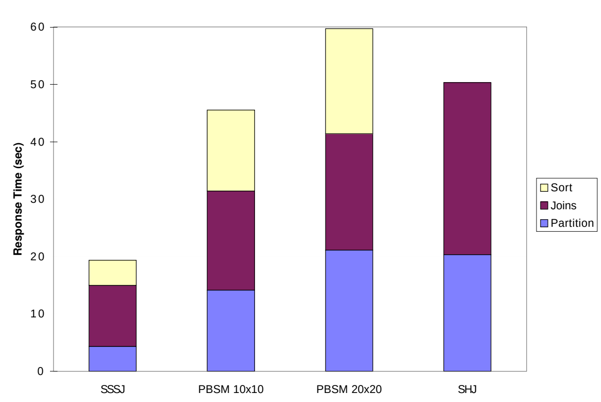
Figure 10: Join performance for uniformly distributed data sets of squares

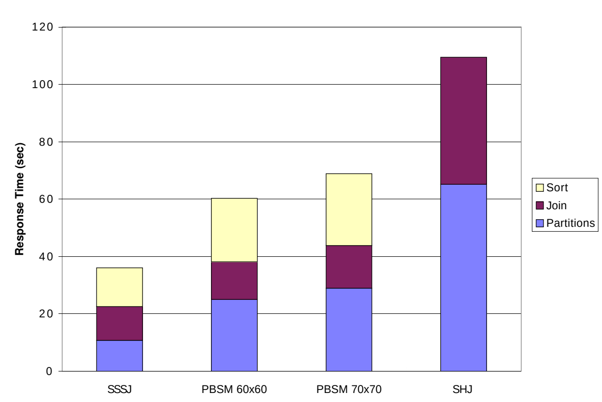




(a)LB and LB' join (b) MG and MG' join.

Figure 11: Join performance for real data sets





(a) Triangular distribution, self join (b) CFD data set self join

Figure 12: Self Join performance for real data sets

the performance of the algorithms on data sets with high coverage, with varying sizes in the spatial entities, and with distributions with high clustering.

Figure 12a presents the results of a self join of TR. Although only a single data set is involved, the algorithm does not exploit that fact. S^3J, with Hilbert value computation, is processor bound. Due to the high coverage in the data set, S^3J has to keep the pages of level files in memory longer while testing for intersections.

PBSM spends most of its time partitioning and joining corresponding partitions but sorting and duplicate elimination also account for a large fraction of the execution time, since the size of the join result is large. In contrast with S^3J, PBSM appears I/O bound.

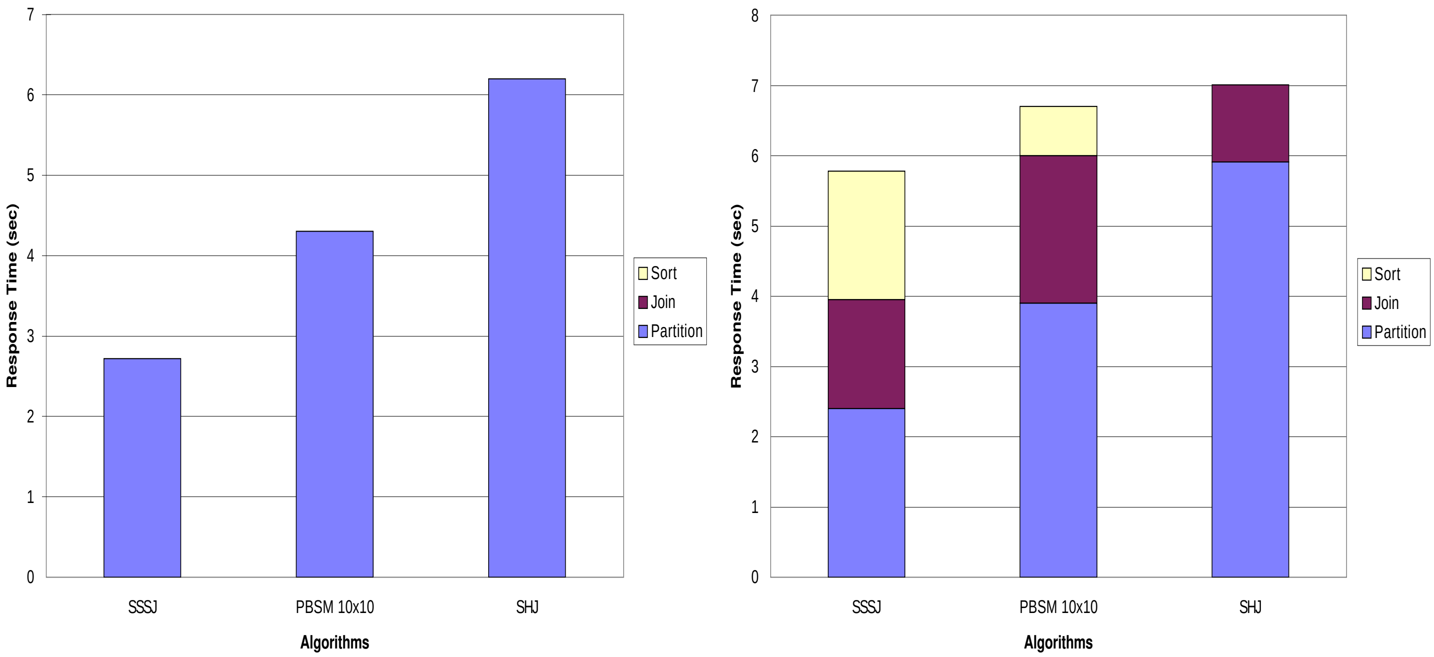
SHJ requires extensive replication during the partitioning of the second data set. This results from the spatial characteristics of the data set and the large number of partitions used. Large variability in the sizes of the entities leads to large partitions. As a result, the probability that an entity will overlap more than one partition increases with the variability of the sizes of the spatial entities. SHJ is I/O bound and most of its time is spent joining pairs of partitions which, in this case, do not fit in main memory. Due to the replication, the time spent by the algorithm partitioning the second data set is much larger than the time spent during the partitioning of the first data set. Although SHJ introduces more replication than PBSM, it does not require duplicate elimination and, depending on the amount of replication and repartitioning performed by PBSM, its partitioning phase might be cheaper. It is due to the fact that no duplicate elimination is needed that SHJ is able to outperform PBSM in the case of large tiles.

Figure 12b presents results from a self join of CFD. We employ a spatial join to find all pairs of points within 10^{-6} distance from each other. For this data distribution, which involves a large cluster in the center of the data space, both PBSM and SHJ perform poorly. PBSM requires a large number of tiles to achieve load balancing for its partitions and a lot of repartitioning takes place, introducing a large degree of replication. The join phase is faster than SHJ however in this experiment since all pairs of partitions obtained via repartitioning fit in main memory. The sampling performed by SHJ is ineffective in this case and the join phase is costly involving a large number of page reads from the disk. The partitions have varying sizes and one of them contains almost the entire data set.

5.2.2 The Effects of Filtering

With the experiments described in the previous subsection, we investigated the relative performance of the algorithms when no filtering takes place during the join of the data sets involved. With the experiments in this section, we study the effects of filtering on all three algorithms.

For this purpose we perform two experiments. In the first we join two uniformly distributed data sets of 50,000 entities each. The data sets are uniformly distributed in disjoint portions of the unit square,



(a) Uniform Data no join case

(b) Uniform Data small join result.

Figure 13: Join performance with filtering for uniformly distributed data sets of squares

so that there is no overlap between the total MBRs of the two data sets. The results are presented in figure 13a.

S^3J incurs a number of disk accesses while reading and partitioning the first data set and requires processor time to create the bitmap for it. During processing of the second data set, the algorithm has to read the data set, but no entities are written back on disk since the complete data set is filtered out with the use of the bitmap constructed during the partitioning of the first data set. PBSM reads the first data set and introduces a 4% replication factor ($r_A = 1.04$) during its partitioning. The second data set is completely filtered out for PBSM as well. PBSM incurs a larger number of I/Os than S^3J. SHJ has to go through its partitioning phase for the first data set. This involves sampling the first data set and populating its partitions. The second data set is filtered out completely by SHJ as well. However, the expensive partitioning phase causes its performance to be worse relative to PBSM in this case.

Figure 13b presents the results of a second experiment involving uniformly distributed squares, but this time, the total MBR of the first data set is contained in the total MBR of the second. The ratio of the area of the total MBR of the two data sets, is one thousand. There exist some joining entities between the two data sets. S^3J reduces the size of the second data set using the bitmap and the total number of page reads and writes of its partitioning, sorting and joining phase are small. PBSM is able to nearly match S^3J's performance since a small replication factor is introduced during the partitioning of the first data set, and a large portion of the second data set is successfully filtered out and not involved in further processing. For SHJ, although a significant amount of filtering takes place also, the expensive partitioning phase once again causes poor performance.

5.3 Discussion

We have presented several experiments comparing the performance of the three algorithms S^3J, PBSM, and SHJ, involving real and synthetic data sets. Our experimental results are consistent with our analytic observations. The relative performance of the algorithms depends heavily on the statistical characteristics of the datasets. S^3J appears to have comparable performance to SHJ when the replication introduced is not large, but is able to outperform it by large factors as replication increases. PBSM is comparable with S^3J when sufficient filtering takes place and in this case performs better than SHJ. The amount of filtering that makes PBSM competitive is difficult to quantify, because it depends on the characteristics of the data sets involved, the amount of replication that PBSM introduces, the order in which the data sets are partitioned, and the number of page reads and writes of the sorting phase of PBSM.

While S^3J neither requires nor uses statistical knowledge of the data sets, the best choice for the number of tiles in PBSM or for the amount of sampling in SHJ depends on the spatial characteristics of the data sets involved in the join operation. Good choices can be made only when statistical information about the data sets is available and the MBRs of the spaces are known. Under uniform distributions, the amount of overlap between the MBRs of the two spaces gives a good estimate of the expected size of the join result. Under skewed data distributions however, no reliable estimate can be made, unless detailed statistical characteristics of both data sets are available. We believe that such measures could be computed for base spatial data sets. However, for intermediate results, the number of page reads required to obtain the statistical characteristics might be high.

It appears from our experiments that, although the partitioning phase of SHJ is expensive, it is worthwhile in the case of low selectivity joins, because it yields a large number of partitions which usually fit in main memory in the subsequent join phase. In contrast, the analytical estimate for the number of partitions of PBSM doesn't consistently yield appropriate values. The partition pairs often do not fit in main memory because of the replication introduced by the algorithm, and the cost of repartitioning can be high.

We experimentally showed that there are data distributions (such as the triangular data distribution we experimented with) for which both PBSM and SHJ are very inefficient. For such distributions it is possible that due to the high replication introduced by both PBSM and SHJ the disk space used for storing the replicated partitions as well as the output of the join before the duplicate elimination in the case of PBSM, is exhausted, especially in environments with limited disk space.

Depending on the statistical characteristics of the data sets involved, S^3J can be either I/O bound or processor bound. We experimentally showed that, even with distributions with many joining pairs, both PBSM and SHJ are I/O bound, but S^3J can complete the join with a minimal number of I/Os and can outperform both other algorithms. For distributions in which filtering takes place, we experimentally showed that S^3J with DSB is able to outperform both PBSM and SHJ. When enough filtering takes place, PBSM does better than SHJ mainly due to the expensive partitioning phase of SHJ. However, the previous argument depends also on the number of tiles used by PBSM, since it might be the case that excessive replication is introduced by PBSM using too many tiles and the performance advantages are lost. S^3J is equally capable of reducing the size of the data sets involved and is able to perform better than both PBSM and SHJ.

6 Conclusions

We have presented a new algorithm to perform the join of spatial data sets when indices do not exist for them. Size Separation Spatial Join imposes a dynamic hierarchical decomposition of the space and permits an efficient joining phase. Moreover, our algorithm reuses software modules and techniques commonly present in any relational system, thus reducing the amount of software development needed to incorporate it. The Dynamic Spatial Bitmap feature of S^3J can be implemented using bitmap indexing techniques already available in most relational systems. Our approach shows that often the efficient bitmap query processing algorithms already introduced for relational data can be equally well applied to spatial data types using our algorithm.

We have presented an analytical and experimental comparison of S^3J with two previously proposed algorithms for computing spatial joins when indices do not exist for the data sets involved. Using a combination of analytical techniques and experimentation with real and synthetic data sets, we showed that S^3J outperforms current alternative methods for a variety of types of spatial data sets.

Several directions for future work arise from this study. First, extension of the algorithms to allow higher dimensions is important for applications like multimedia query processing where descriptions of multimedia objects are represented as points in a multidimensional space [Fal96]. Secondly, parallelism is necessary for high performance query processing on non traditional data types. Exploration of efficient parallelization of previously proposed algorithms as well as S^3J, for low and high spatial dimensions is an interesting and important area for future study.

7 Acknowledgments

We thank Dave DeWitt, Ming Ling Lo, Jignesh Patel and Chinya Ravishankar for their comments and clarifications of the operation of their respective algorithms. We would also like to thank Al Cameau of the IBM Toronto Laboratory for useful discussions regarding our implementations, and Scott Leutenegger of the University of Denver for making the CFD data set available to us. This research is being supported by the Natural Sciences and Engineering Council of Canada, Information Technology Research Centre of Ontario and the IBM Toronto Laboratory.

References

[Bia69] T. Bially. Space-Filling Curves: Their Generation and Their Application to Bandwidth Reduction. *IEEE Trans. on Information Theory*, IT-15(6):658–664, November 1969.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient Processing of Spatial Joins using R-trees. *Proceedings of ACM SIGMOD*, pages 237–246, May 1993.

[BKSS94] Thomas Brinkhoff, H.P Kriegel, Ralf Schneider, and Bernhard Seeger. Multistep Processing of Spatial Joins. *Proceedings of ACM SIGMOD*, pages 189–208, May 1994.

[Bur91] Bureau of the Census. TIGER/Line Census Files. March 1991.

[Fal96] Christos Faloutsos. *Indexing Multimedia Databases.* Kluwer, September 1996.

[Gun93] Oliver Gunther. Efficient Computation of Spatial Joins. *Proceedings of the International Conference on Data Engineering.*, pages 50–59, April 1993.

[Gut84] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. *Proceedings of ACM SIGMOD*, pages 47–57, June 1984.

[LR94] Ming-Ling Lo and Chinya. V. Ravishankar. Spatial Joins using Seeded trees. *Proceedings of ACM SIGMOD*, pages 209–220, May 1994.

[LR95] Ming-Ling Lo and Chinya V. Ravishankar. Generating Seeded Trees from Spatial Data Sets. *Symposium on Large Spatial Data Bases*, pages 328–347, August 1995.

[LR96] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. *Proceedings of ACM SIGMOD*, pages 247–258, June 1996.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM TODS 1984*, pages 38–71, May 1984.

[OG95] P. O'Neil and G. Graefe. Multi-Table Joins Through Bitmapped Join Indeces. *SIGMOD Record Vol. 24, No. 3*, pages 8–11, September 1995.

[O'N96] P. O'Neil. Query Performance. *Talk Delivered at IBM Toronto*, March 1996.

[Ore86] J. Orenstein. Spatial Query Processing in an Object-Oriented Database System. *Procedings of ACM SIGMOD*, pages 326–336, May 1986.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition Based Spatial-Merge Join. *Proceedings of ACM SIGMOD*, pages 259–270, June 1996.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York-Heidelberg-Berlin, October 1985.

[Rot93] Doron Rotem. Spatial Join Indices. *Proceedings of the International Conference on Data Engineering*, pages 500–509, March 1993.

[SK96] Kenneth C. Sevcik and Nick Koudas. Filter Trees for Managing Spatial Data Over a Range of Size Granularities. *Proceedings of VLDB*, pages 16–27, September 1996.

[SM96] M. Stonebraker and D. Moore. *Object Relational Databases: The Next Wave.* Morgan Kauffman, June 1996.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+ -tree : A Dynamic Index for Multi-dimensional Data. *Proceedings of VLDB 1987*, pages 507–518, September 1987.

[Val87] P. Valduriez. Join Indexes. *ACM TODS, Volume 12, No 2*, pages 218–246, June 1987.

Contents