A Non-blocking Parallel Spatial Join Algorithm

Gang Luo Jeffrey F. Naughton
Department of Computer Sciences
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53705
{gangluo, naughton}@cs.wisc.edu

Curt J. Ellmann
NCR Advance Development Lab
5752 Tokay Boulevard, Suite 400
Madison, WI 53719
Curt.Ellmann@ncr.com

Abstract

Interest in incremental and adaptive query processing has led to the investigation of equijoin evaluation algorithms that are non-blocking. This investigation has yielded a number of algorithms, including the symmetric hash join, the XJoin, the Ripple Join, and their variants. However, to our knowledge no one has proposed a non-blocking spatial join algorithm. In this paper, we propose a parallel non-blocking spatial join algorithm that uses duplicate avoidance rather than duplicate elimination. Results from a prototype implementation in a commercial parallel object-relational DBMS show that it generates answer tuples steadily even in the presence of memory overflow, and that its rate of producing answer tuples scales with the number of processors. Also, when allowed to run to completion, its performance is comparable with the state-of-the-art blocking parallel spatial join algorithm.

1. Introduction

Recently, there has been a lot of attention in the research literature on non-blocking query evaluation. This work has found application in adaptive query processing [9], in dealing with unpredictable rates of input data sources [13, 21], and in online aggregation and online data visualization [11, 10, 7]. To date, most work has focused on the implementation of non-blocking equijoin algorithms, and, to the best of our knowledge, there is no published non-blocking spatial join algorithm. Without such an algorithm, queries involving spatial joins remain "off limits" for non-blocking evaluation techniques and applications. In this paper we present a parallel non-blocking spatial join algorithm.

Non-blocking spatial operators are likely to be especially useful in applications like online data visualization [8, 7]. Such applications involve a user who wishes to interactively explore a large data set that involves spatial attributes. Complex queries over such data sets often take a long time to execute, forcing the user into a much less productive batch-oriented style of interaction with the data.

One such application is the Sloan Digital Sky Survey (SDSS), recently reported in [19]. In that application, a query might be: "find objects within 10 arcseconds of each other that have identical colors, but have a different brightness". This involves a spatial join over multiple terabytes of data. If a user is forced to wait for the query to execute to completion before seeing results, batch operation is inevitable. By contrast, online spatial operators could allow the user to begin seeing results immediately, perhaps allowing them to abort or refine the query long before the batch execution would have completed. Furthermore, on data sets of this size, the non-blocking operator must be able to use parallelism and the power of multiprocessors if the user is to be able to see a substantial number of answers in a reasonable amount of time.

Designing a non-blocking spatial join operator presents challenges not found when designing non-blocking equijoin operators. One reason for this is that spatial joins are more complex than equijoins, both because the join predicate, polygon overlap, is more complex than equality, and because spatial objects are generally larger than the relational tuples handled in equijoin applications. Another reason for this added complexity is that spatial partitioning necessarily introduces replication, since objects that span partitions must be represented in each partition. This in turn raises the possibility of spurious duplicate generation. These duplicates pose a problem with respect to building a non-blocking spatial join operator, since the required duplicate eliminating post-processing step is inherently blocking. One could consider implementing a non-blocking duplicate elimination operator, but this would require comparing every answer tuple with every previously generated answer, which will be inefficient if the result is large (especially if it is larger than memory).

In our work we deal with this issue by adapting the approach recently introduced in [3] – instead of duplicate elimination, we use duplicate avoidance. In [3], the authors considered duplicate avoidance in the context of a blocking uniprocessor spatial join; in our work, in addition to showing that duplicate avoidance is useful in the context of non-blocking algorithms, we show that duplicate avoidance vs. duplicate elimination exhibits interesting tradeoffs in a parallel environment.

In our experiments with an implementation in the TOR parallel object-relational DBMS (ORDBMS), we provide a performance study of duplicate avoidance vs. duplicate elimination in a parallel environment. In addition, we show that the non-blocking parallel spatial join algorithm scales well (when allowed to run to completion, its performance is comparable to that of the state-of-the-art blocking parallel spatial join algorithm), and steadily generates join result tuples at a rate roughly linear in the number of processors.

2. Non-blocking Parallel Spatial Join Algorithm

We begin our discussion of our non-blocking parallel spatial join algorithm by reviewing previously proposed algorithms for equijoins. Non-blocking equijoin algorithms are based upon the symmetric hash join originally proposed in the PRISMA parallel database project [5]. Briefly, unlike the blocking hash join, the symmetric hash join builds hash tables on both inputs. Suppose we are using this algorithm to perform the equijoin of two relations *R* and *S*. Then when a tuple *r* from *R* is processed, it is first used to probe the hash table on *S* and then inserted into the hash table for *R*. Similarly, when a tuple *s* from *S* arrives, *s* is first used to probe the hash table on *R*, then inserted into the hash table for *S*. A moment's thought shows that this algorithm produces each join result tuple exactly once.

For spatial overlap joins, hash indices are not useful. Instead, we use R-trees [6, 1, 4]. At the simplest level, our non-blocking spatial join algorithm looks just like the symmetric hash equijoin, except that we use R-trees instead of hash tables, and the "probe" operator looks for overlapping tuples instead of exact matches. However, if we simply used R-trees and let them overflow to disk when they grow larger than main memory, performance would not be acceptable, since in the limit every R-tree lookup would result in some number of disk I/Os. Accordingly, we explicitly and carefully manage memory so that performance is not sacrificed yet the online, steady production of join result tuples is maintained, even in the presence of memory overflow. The following subsections give more detail as to how our algorithm incorporates these memory management techniques. We return to the orthogonal issue duplicate avoidance in Section 3.

2.1. Overview of the Non-blocking Parallel Spatial Join Algorithm

Suppose we want to join two relations *A* and *B* on the spatial attributes *A.c* and *B.d* as in the following extended SQL query:

```
select online *
from A, B
where A.c overlaps B.d;
```

Here, by "online" we mean that the query should produce the tuples continuously as they are generated [11, 10] (rather than producing the full result at once after all processing is complete).

Our target machine for this join is a shared-nothing multiprocessor [2]. We refer to the processor-disk units in the system as "data server nodes." If the actual hardware platform is a cluster of SMPs, it is possible that several data server nodes could be grouped together on the same symmetric multiprocessor (SMP). From a software viewpoint, we do not distinguish between data server nodes on the same SMP and data server nodes on different SMPs.

We assume that before beginning the join, the tuples of *A* and *B* are distributed about the data server nodes according to some partitioning strategy that may have nothing to do with the spatial join attributes. Because of this, the first step of the parallel spatial join is to redistribute the tuples of *A* and *B* according to a spatial partitioning. The goal of this redistribution is to distribute the tuples of the join relations so that each node performs roughly equal work during the execution of the algorithm. This is done with the help of a spatial partitioning function as is described below.

We define the universe as the minimum rectangle that covers the spatial join attribute values for all tuples of the two join relations *A* and *B*. This information is usually part of the statistics stored in the database catalogs. Suppose there are a total of *L* data server nodes in addition to a query coordinator node. We decompose the universe into *L* subparts using a spatial partitioning function *SPF*, where each subpart is mapped to a node. We call the portion of the universe mapped to a node a "partition." For our spatial partitioning function we use the tile method [14]. The tile method works as follows. First, we divide the universe into N_T equal sized rectangular tiles, where $N_T \geq L$. Each tile is mapped to a partition using a hash function. ([14] showed that hashing is superior to round robin when allocating tiles to partitions.) For example, consider Figure 1 where the universe is divided into 6 tiles, the number of partitions is 3, and tiles are mapped to partitions using a round robin scheme. Thus tiles 0 and 3 are mapped to partition 0, tiles 1 and 4 are

mapped to partition 1, and tiles 2 and 5 are mapped to partition 2.

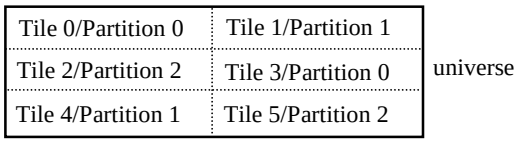


Figure 1. A spatial partitioning function using tiles.

Figure 2 is an example where the universe is partitioned into four subparts. In this example we assume that there is a one-to-one mapping from tiles to partitions, so we omit any mention of tiles.

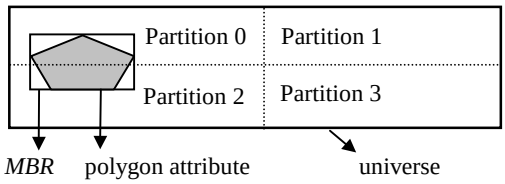


Figure 2. Spatially partitioning the universe into four subparts.

For each tuple T_A of A or T_B of B, we check the minimum bounding rectangle (*MBR*) of its join attribute value to determine all the partitions with which the *MBR* overlaps. Then we redistribute this tuple to all the corresponding nodes. Continuing with our example, in Figure 2, the polygon is redistributed to node 0 and node 2. Note that we need to check the *MBR* to determine all the tiles with which the *MBR* overlaps. Thus, if the *MBR* overlaps with tiles from multiple partitions, we will redistribute the tuple to all nodes according to these partitions. As we mentioned in the introduction, a join result tuple can be generated multiple times in different partitions. We will describe our duplicate avoidance techniques in Section 3.

For the non-blocking parallel spatial join algorithm, we do the following three things simultaneously by multi-threading at each node:

1. Thread *RedisA* redistributes the tuples of A to all nodes according to the spatial partitioning function *SPF*.
2. Thread *RedisB* redistributes the tuples of B to all nodes according to the spatial partitioning function *SPF*.
3. Thread *JoinAB* performs the local join of the incoming tuples of A and B from redistribution using the adaptive symmetric spatial join algorithm described below.

2.2. Adaptive Symmetric Spatial Join Algorithm

At each node, we maintain two in-memory R-trees: RT_A for A, and RT_B for B. Their search keys are the spatial join attributes. We also maintain two disk-resident bucket tables: BT_A for A, and BT_B for B. Each bucket table contains N buckets. We partition the subpart SP of the universe at the node into N partitions using another spatial partitioning function SPF'. Each bucket B_A of BT_A (B_B of BT_B) corresponds to a partition of SP, and vice versa. We call each pair of buckets, B_A of BT_A and B_B of BT_B, which correspond to the same partition of SP, the bucket pair B_{AB}. For performance reasons, we keep a write buffer P_A (P_B) of one page in memory for each bucket B_A (B_B). We can think of all buckets in BT_A (BT_B) as sharing the same memory part RT_A (RT_B). Figure 3 shows the data structures at a specific node.

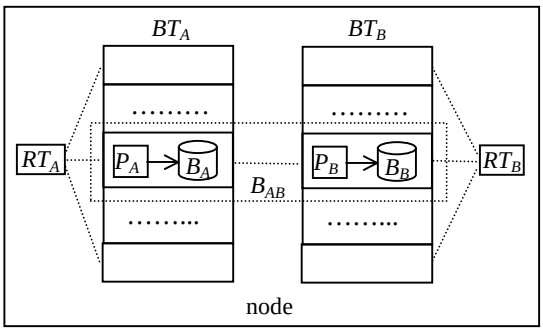


Figure 3. Data Structures at a node.

The adaptive symmetric spatial join algorithm at each node is composed of two stages.

2.2.1. First Stage: Redistribution Phase. The goal of the first stage is to redistribute and join as many of the tuples of A and B as possible with the available memory. The first stage is complete when all the tuples of both relations have been redistributed. Note that answer tuples are produced during both stages of the algorithm. The first stage runs until all the tuples have been redistributed, generating answer tuples as well as potentially spooling some tuples to disk; the second stage "cleans up" by joining tuples that have been spooled to disk in the first phase.

Stage 1-1: memory-resident redistribution phase

When a tuple T_A of A or T_B of B comes from redistribution,

if the tuple is T_A,

T_A is inserted into RT_A and joined with the tuples in RT_B.

If memory becomes full, we enter stage 1-2.

If the tuple is T_B, we do the same operations described above except we switch the positions of A and B.

Stage 1-2: memory overflow redistribution phase

When a tuple T_A of A or T_B of B comes from redistribution,

if the tuple is T_A,

T_A is joined with the tuples in RT_B.

We check the *MBR* of T_A's join attribute value to determine all the partitions of *SP* with which the *MBR* overlaps, using the spatial partitioning function *SPF*′. T_A is inserted into all the B_A buckets corresponding to these partitions in the following way: for each such B_A, we write T_A to the write buffer P_A. Whenever P_A becomes full, we write P_A to DP_A, then P_A can accept tuples from *A* again.

If the tuple is T_B, we do the same operations described above except we switch the positions of *A* and *B*.

As a special case, by the time memory becomes full at stage 1-1, if all tuples of *B* (*A*) have arrived from redistribution, BT_B (BT_A) will be empty. At stage 1-2, whenever a tuple T_A of *A* (T_B of *B*) comes from redistribution, we only need to join it with the appropriate tuples in RT_B (RT_A) without writing it to BT_A (BT_B). In this way, we avoid the work of the second stage.

The representation of a spatial object can be very large. This poses a problem because we cannot insert a spatial object whose size is larger than a page into an in-memory R-tree directly. For example, a spatial object representing a polygon might require thousands of points to represent the exact geometric shape. For this reason, when we insert a tuple with some spatial attribute into an in-memory R-tree, we store the tuple in memory and insert the memory address (i.e., pointer) of the tuple into the R-tree instead of the tuple itself.

2.2.2. Second Stage: Disk Reread Phase. When all the tuples of *A* and *B* have arrived from redistribution, we free RT_A and RT_B, and enter the second stage for the node. At this time, for a bucket pair B_{AB}, all tuples in B_B (B_A) have been joined with the tuples in RT_A (RT_B) and all tuples in RT_A have been joined with the tuples in RT_B. We only need to join the tuples in B_A with the tuples in B_B.

In a typical application of non-blocking query evaluation, such as online data visualization, users may be still looking for an approximate answer upon memory overflow. To partially fulfill this requirement, we may generate the join result tuples in a nearly random order at the second stage. To do this heuristically, we use a random selection algorithm to select the bucket pairs one by one, randomly and non-repetitively.

Organize all the bucket pairs into an array *HT*.

While (number of elements in *HT* > $const_N$)

Randomly select ($const_P$ × number of elements in *HT*) elements in *HT*,

if this element in *HT* has not been selected before, select it.

Compact *HT* so that it only contains the elements that have not been selected before.

While (*HT* is not empty)

Randomly select an element in *HT*.

Compact *HT* so that it only contains the elements that have not been selected before.

We should choose the constants $const_N$ and $const_P$ to be small numbers, such as 10 and 15%, respectively.

For each bucket pair, we do the following operations:

Initialize two in-memory R-trees RT_A′ and RT_B′.

Read the tuples in B_A (including the tuples in P_A) and tuples in B_B (including the tuples in P_B) from the disk into memory simultaneously, and for each tuple immediately make the following decision:

if the tuple is T_A, T_A is inserted into RT_A′ and joined with the tuples in RT_B′.

If the tuple is T_B, T_B is inserted into RT_B′ and joined with the tuples in RT_A′.

Free RT_A′ and RT_B′.

2.2.3. Work Done at Different Stages. For a bucket pair B_{AB}, the stages at which the work is done for spatially joining tuples of *A* and tuples of *B* are shown in Figure 4 and Figure 5:

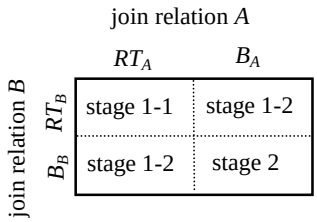
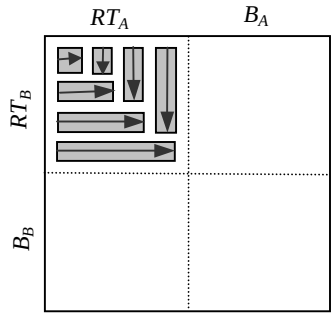
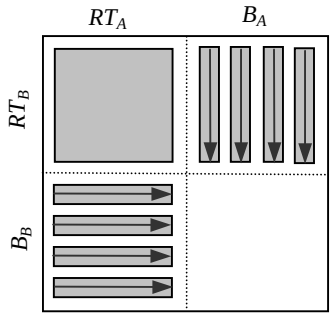


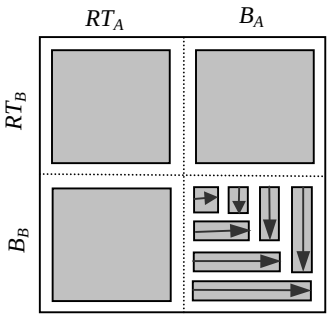
Figure 4. Work done at each stage for a bucket pair (global view).



(1) Stage 1-1



(2) Stage 1-2



(3) Stage 2

Figure 5. Work done at each stage for a bucket pair (details).

2.3. Handling Partition Overflow

If the data is very skewed or the data set is very large, it is possible for the adaptive symmetric spatial join algorithm to have bucket pairs that do not fit entirely in memory at the second stage. One way to handle this is to increase the number of bucket pairs (limited to *M/2*) at the beginning. Another alternative is to modify the second stage. For each bucket pair that cannot fit in memory, we treat the tuples in the bucket pair as if they just come from redistribution and do an adaptive symmetric spatial join on them recursively.

2.4. Determining the Number of Bucket Pairs

In order to avoid partition overflow, the total memory requirement of the non-blocking parallel spatial join algorithm is the larger of the combined size of RT_A, RT_B, and all the write buffers (P_A and P_B), and the combined size of RT_A' and RT_B'. The combined size of RT_A' and RT_B' is roughly equal to the size of the largest disk-resident bucket pair B_{AB}.

Suppose there are a total of L data server nodes, and all the tuples are evenly redistributed among these nodes. Also, assume that on average, each tuple is redistributed to p data server nodes. Assume there are H bucket pairs at the node, and all the tuples coming from redistribution are evenly distributed among the bucket pairs. Furthermore, suppose that on average, each tuple is replicated at p' bucket pairs. Notice that p and p' depend on L and H, respectively. Let $||x||$ denote the size of x in pages, M denote the size of available memory in pages, and A' and B' denote the tuples of A and B that are redistributed to this node, respectively. $\|A'\| \approx \|A\| p / L$, $\|B'\| \approx \|B\| p / L$, $\left\|RT_A'\right\| + \left\|RT_B'\right\| \approx \|B_{AB}\| \approx (\|A'\| + \|B'\|) p' / H \approx (\|A\| + \|B\|) pp'/(HL)$, $\|P_A\| \geq 1$, and $\|P_B\| \geq 1$.

$$M \geq \max(\|RT_A\| + \|RT_B\| + H(\|P_A\| + \|P_B\|), \left\|RT_A'\right\| + \left\|RT_B'\right\|)$$

$$\geq \max(\|RT_A\| + \|RT_B\| + 2H, (\|A\| + \|B\|) pp'/(HL)) \geq (\|A\| + \|B\|) pp'/(HL).$$

Thus in order to avoid partition overflow, the non-blocking parallel spatial join algorithm requires the following condition limiting the number of bucket pairs be satisfied: $H \geq (\|A\| + \|B\|) pp'/(LM)$.

For example, suppose that we have a 32-node multi-processor system, and that each processor has 1GB memory. Furthermore, assume that H=1000, p=1.5, and p'=5 (these values are based on the measurements from the Sequoia data sets [18].) Then we can process a 4 TB data set without requiring any partition overflow handling techniques.

2.5. Spatial Distance Join Extension

The non-blocking parallel spatial join algorithm can be easily extended to support spatial distance join [12, 20]. A typical SQL statement that uses distance join is

select online *
from *A*, *B*
where *distance(A.c, B.d)* < *D*;

For the join attribute value of each tuple T_A of A (T_B of B), as proposed in [17], we can expand the four sides of its *MBR* by *D/2* as shown in Figure 6. For two tuples T_A of A and T_B of B, if *distance*($T_A.c$, $T_B.d$) < D, the expanded bounding rectangles of their join attribute values must overlap. Thus if we use the expanded bounding rectangle instead of *MBR* in the non-blocking parallel spatial join algorithm, we can find all the tuple pairs (T_A, T_B) that are within distance D.

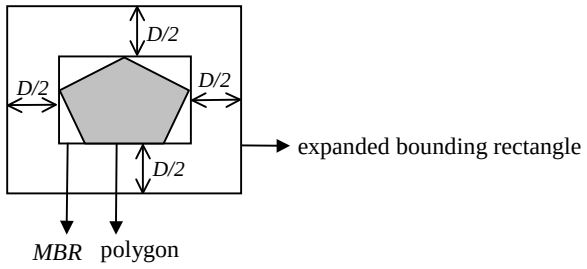


Figure 6. Expanding bounding rectangle.

3. Duplicate Avoidance Techniques

We now describe the techniques we use to avoid duplicates in the course of the join. In the non-blocking parallel spatial join algorithm, the join result tuple of T_A and T_B may be generated multiple times. These duplicates arise in two ways:

1. If both T_A and T_B are replicated to several nodes by redistribution, duplicates will be generated at the nodes at which both of them are replicated.

2. At a specific node, if both T_A and T_B are replicated in several bucket pairs during the second stage of the adaptive symmetric spatial join algorithm, duplicates will be generated at the bucket pairs where both of them are replicated. Note that this second source of duplicates is the only one that occurs in uniprocessor systems.

In order to let our parallel spatial join algorithm be non-blocking, we need to avoid generating duplicates for the join result tuples online rather than using a duplicate removal operator at the last step of the algorithm. We introduce the duplicate avoidance techniques in the following two sections.

3.1. Revised Reference Point Method

To avoid generating duplicates among different nodes, we use the reference point method in [3]. For each pair of tuples T_A and T_B, if both of them are replicated at several nodes, we only join them at one of the nodes where both of them are replicated. To attempt to give each node a roughly equal workload, we slightly revise the original reference point method. Let M_A and M_B denote the *MBR*s of the spatial join attribute values of tuples T_A and T_B, respectively. Let (xl, yl) and (xh, yh) denote the coordinates of the lower left corner and upper right corner of a *MBR*, respectively.

For a pair of intersecting *MBR*s M_A and M_B, we define four reference points:

$P_0 = (max(M_A.xl, M_B.xl), max(M_A.yl, M_B.yl))$,
$P_1 = (max(M_A.xl, M_B.xl), min(M_A.yh, M_B.yh))$,
$P_2 = (min(M_A.xh, M_B.xh), max(M_A.yl, M_B.yl))$,
$P_3 = (min(M_A.xh, M_B.xh), min(M_A.yh, M_B.yh))$.

As M_A and M_B intersect with each other, each reference point must be within both *MBR*s M_A and M_B. For example, suppose $M_B.xl \le M_A.xl$, then $max(M_A.xl, M_B.xl)=M_A.xl \le M_B.xh$, otherwise M_A and M_B can not intersect with each other. Thus $M_A.xl \le max(M_A.xl, M_B.xl) \le M_A.xh$, $M_B.xl \le max(M_A.xl, M_B.xl) \le M_B.xh$. Other cases can be reasoned in a similar way.

Figure 7 shows the situation where there are four data server nodes, the universe is partitioned into four subparts, and each data server node manages one subpart of the universe.

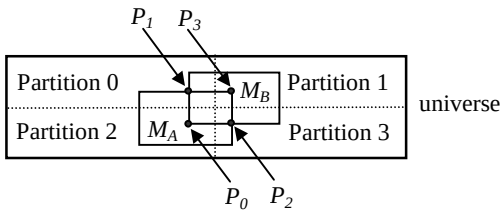


Figure 7. Revised reference point method.

We use $M_A.xl+M_B.xl$ as the seed of a random number generator to generate a random number r. Let $s=r \bmod 4$. Tuples T_A and T_B are only joined at the node that manages the partition containing point P_s. Here, in order to balance the workload on the nodes, we introduce randomness in deciding which node to perform the spatial join operation.

Our approach differs from the original reference point method in [3] in that in the original algorithm, the same reference point is used for each pair of tuples T_A and T_B. While this has proven to be effective in uniprocessor applications, in a parallel system it could cause the workload on the nodes to be unbalanced. For an extreme example, consider the case where all the spatial join attribute values of relations *A* and *B* are nearly of the same size as the universe. If we use the original reference point method without any change, then almost all the spatial join operations will be done at one node. This situation is illustrated in Figure 8 where almost all the spatial join operations are done at the node managing partition 0 of the universe.

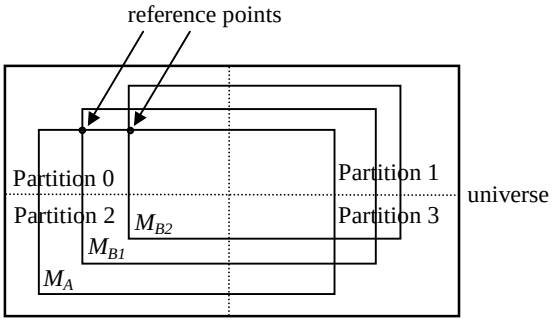


Figure 8. Skew in the original reference point method.

3.2. Avoiding Generating Duplicates among Different Bucket Pairs

To avoid generating duplicates among different bucket pairs, we again use the revised reference point method at the second stage of the adaptive symmetric spatial join algorithm. For each pair of tuples T_A and T_B, if both of them are replicated at several bucket pairs, we only join them at the bucket pair that manages the partition containing the selected reference point.

4. Performance

In this section, we present results from a prototype implementation of the non-blocking parallel spatial join algorithm in the Teradata parallel ORDBMS TOR 2.0.02. Our measurements were performed with the TOR client application and server running on Intel x86 Family 6 Model 5 Stepping 3 workstations each with four 400MHz processors, 1GB main memory, six 8GB disks, running

the Microsoft Windows NT 4.0 operating system. We allocated a processor and a disk for each data server, so there were at most four data servers on each workstation. Although each workstation had 1GB of memory, in our experiments we limited the DBMS to use 1.5 MB memory for each data server node so that we could explore the impact of memory overflow on the performance of our algorithms.

The relations used for the tests were the standard Sequoia Benchmark relations [18]. The polygon relation represents regions of homogeneous land use characteristics in the State of California and Nevada, while the islands relation represents holes in the polygon data (for example, a lake in a park). Their schemas are shown as follows:

```
create table polygon
( landuse integer not null,
  shape closedpolygon not null
);
create table islands
( hole closedpolygon not null
);
```

Table 1 shows the characteristics of the two relations.

Table 1. Sequoia data.

relation	number of tuples	total size	average number of points in a tuple
polygon	58296	23.1MB	49
islands	20976	5.1MB	30

In the experiments, we ran the following query to join the polygons and islands based on the condition of overlapping. No index was created for the attributes.

```
select online *
from polygon, islands
where polygon.shape overlaps islands.hole;
```

The most important performance metric for online algorithms is the rate at which join result tuples are generated.

We ran the query on 1-node, 2-node, 4-node, 8-node, and 16-node configurations (where each node is a data server) with a global query coordinator. In addition to our non-blocking parallel spatial join (NBPS) algorithm, we ran the classical blocking parallel partition based spatial-merge join (PPBSM) algorithm [14, 16, 15] for comparison. We chose the state-of-the-art PPBSM algorithm because PBSM performs well among all the traditional blocking spatial join algorithms when neither join input has a pre-existing index [14], which is the case in our parallel environment because the join inputs are dynamically redistributed. In order to measure the impact of the duplicate avoidance techniques on performance we also ran a modified version of our non-blocking parallel spatial join algorithm. In this version, we deleted the duplicate avoidance techniques and instead added a post-processing blocking duplicate elimination pass. We call this algorithm NBPS-WD, where the WD stands for "Without Duplicate avoidance."

For the NBPS algorithm, Figure 9 shows how the number of join result tuples generated increases over time. Note that the number of tuples generated at any given time increases substantially with the number of processors in the configuration. Thus, the NBPS algorithm meets our goal of using parallelism to increase the rate of production of answer tuples. Also, for all configurations, answers are produced steadily and incrementally, as is required by the definition of non-blocking query evaluation.

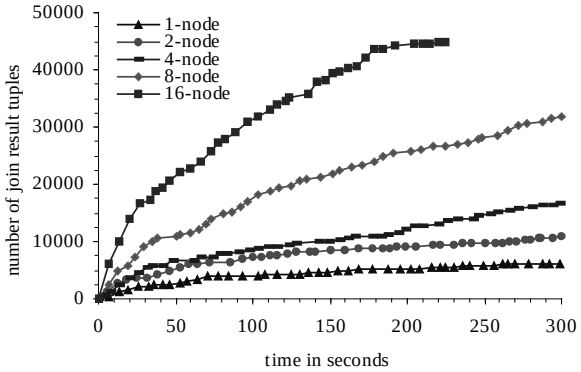


Figure 9. Join result tuples generated over time.

For the NBPS algorithm, Table 2 shows the time at which memory overflow occurred in the five configurations. We can see that memory overflow has only a minimal impact on the speed at which join result tuples are generated. The time at which memory overflows in the various configurations exhibits a somewhat subtle behavior. In each configuration, all nodes fetch tuples from the join relations and do the join work in parallel, so one might think that memory overflow would occur at the same time for all five configurations. In fact, as the number of nodes increases, the spatial range that the join attribute occupies at each node decreases and the local join selectivity at each node increases. Thus in the same amount of time, a larger percentage of the time is spent on joining rather than fetching tuples from the join relations at each node, and memory overflow occurs later.

Table 2. Time (seconds) that memory overflow occurs.

1-node	2-node	4-node	8-node	16-node
8	11	22	23	19

Figure 10 shows the execution time of the NBPS algorithm, the PPBSM algorithm, and the NBPS-WD algorithm in the five configurations when allowed to run to completion. We found that in all five configurations, the NBPS algorithm is faster than the PPBSM algorithm. This is somewhat surprising, since the NBPS algorithm has to build two R-trees "on the fly," and it is usually the case that such an approach cannot match the performance

of an algorithm like PPBSM, which uses more efficient computational geometry techniques in its processing [14]. Upon further investigation, it became clear that the good performance of NBPS was due to its duplicate avoidance techniques. Notice that NBPS without using the duplicate avoidance techniques, NBPS-WD, is 43%~84% slower than NBPS.

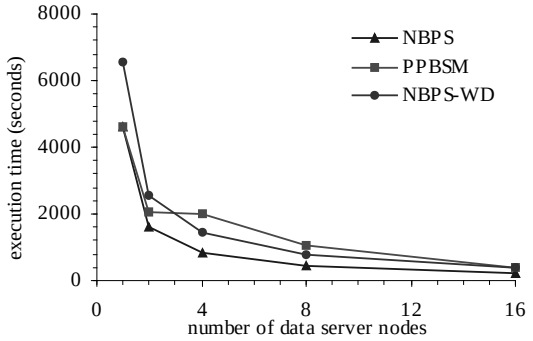


Figure 10. Execution time of the three parallel spatial join algorithms (speedup).

It turns out that the number of duplicates generated in the various configurations varies in an interesting manner with the number of nodes in the configuration. For the NBPS-WD algorithm, Table 3 shows the ratio of number of duplicates to number of join result tuples in percentage. We can see that 10%~34% join result tuples are generated as duplicates and need to be eliminated by a duplicate removal operator at the last step of the NBPS-WD algorithm. Thus, the duplicate avoidance techniques are very effective. In 1-node and 2-node configurations, since there are only a few data server nodes, only a few duplicates are generated, and these are mainly from different bucket pairs within the nodes. In the 4-node and 8-node configurations, we see an increase in the number of duplicates, as more duplicates are generated between different nodes. However, in the 16-node configuration, the number of duplicates begins to drop. This is because with more data server nodes, fewer tuples are redistributed to each data server node, so more tuples can be held in the in-memory R-trees. This in turn means that they are not written into the bucket pairs. Thus fewer duplicates are generated between the bucket pairs, which causes fewer duplicates to be generated in total.

Table 3. Ratio of duplicates to join result tuples in percentage for NBPS-WD.

1-node	2-node	4-node	8-node	16-node
10%	13%	34%	30%	22%

Perhaps even more surprisingly, although in the 1-node and 2-node configurations NBPS-WD is slower than the PPBSM algorithm, in the 4-node, 8-node, and 16-node configurations the NBPS-WD algorithm is faster than PPBSM. The reason for this once again has to do with duplicates. Recall that the NBPS-WD algorithm keeps some fraction of the input tuples in the in-memory R-trees, never writing them to disk partitions. This fraction increases with the number of processors, since the aggregate memory of the system increases with the number of processors. PPBSM, on the other hand, always writes all tuples to disk partitions at some point. One might think then that the better performance of NBPS-WD over PPBSM is due to reduced I/O. This is not the case; both of these algorithms are wildly CPU-bound.

To see just how CPU bound these joins are, consider the following calculation. Assume that in the 1-node configuration, for the NBPS-WD algorithm, each tuple is replicated to 1.5 bucket pairs on average (this is an over estimate). Then on average, each tuple is read from the disk at most 2.5 times, and written to the disk at most 1.5 times. Since the total size of the two input relations is less than 30 MB, the total I/O is less than 120MB. Even assuming a very slow I/O rate, say 3MB/second, I/O is still less than 1% of the total execution time.

So the superior performance of the NBPS-WD algorithm over PPBSM is due to a more subtle effect. As we have mentioned before, the NBPS-WD algorithm can only generate duplicates within a node (among different bucket pairs) for tuples that are written to disk partitions. Since the NBPS-WD algorithm keeps more tuples in memory for the configurations with more processors, it generates fewer duplicates locally among different bucket pairs, and therefore avoids the CPU cost of generating those duplicate join result tuples. This trend can be seen from Table 4, which shows the total number of duplicates eliminated (both among different nodes and among different bucket pairs) for the PPBSM algorithm and the NBPS-WD algorithm.

Table 4. Total number of duplicates eliminated for PPBSM and NBPS-WD.

	1-node	2-node	4-node	8-node	16-node
PPBSM	8939	12467	26489	28337	30945
NBPS-WD	4563	6057	15119	13365	9957

In [3], the authors also observed that duplicate avoidance performed better than duplicate elimination, although since they considered only a uniprocessor environment they did not observe the affect that the number of processors has on the relative performance of duplicate elimination vs. duplicate avoidance.

Both the NBPS and PPBSM algorithms take from hundreds to thousands of seconds to run to completion. However, the NBPS algorithm has better speedup than the PPBSM algorithm. Moreover, the NBPS algorithm starts to generate join result tuples continuously and steadily within seconds, while the PPBSM algorithm produces no result until near completion, which is two to three orders of magnitude slower.

The non-blocking parallel spatial join algorithm has good speedup and the duplicate avoidance techniques work better with more nodes. Thus, the performance advantage of the non-blocking parallel spatial join algorithm would increase if either the sizes of the join relations (data size) or the number of nodes (degree of parallelism) were increased.

5. Conclusions

This paper proposes the first non-blocking parallel spatial join algorithm, allowing spatial join processing to participate in the increasingly important domain of non-blocking or adaptive query processing. We introduce the revised reference point method to avoid generating duplicates, which avoids executing the blocking and time-consuming duplicate removal operator, and show how memory can be explicitly managed so as to maintain a steady stream of result tuples even in the presence of memory overflow. Our implementation in a commercial parallel ORDBMS shows that the non-blocking parallel spatial join algorithm has good speedup properties and, when allowed to run to completion, comparable performance with the state-of-the-art blocking parallel partition based spatial-merge join algorithm.

Acknowledgements

We would like to thank Josef Burger and Kevin O'Connor for useful discussions. This work was supported by the NCR Corporation and also by NSF grants CDA-9623632 and ITR 0086002.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conf. 1990: 322-331.

[2] D.J. DeWitt, J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. CACM 35(6): 85-98, 1992.

[3] J. Dittrich, B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. ICDE 2000: 535-546.

[4] V. Gaede, O. Günther. Multidimensional Access Methods. Computing Surveys 30(2), 1998: 170-231.

[5] G. Graefe. Query Evaluation Techniques for Large Databases. ACM Comput. Surveys, 25(2):73-170, June 1993.

[6] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conf. 1984: 47-57.

[7] P.J. Haas. Techniques for Online Exploration of Large Object-Relational Datasets. Proc. 11th Intl. Conf. Scientific and Statistical Database Management, 1999, 4-12.

[8] J.M. Hellerstein, R. Avnur, and A. Chou et. al.. Interactive Data Analysis: The CONTROL Project. IEEE Computer, 32, August 1999, 51-59.

[9] J.M. Hellerstein, M. Franklin, and S. Chandrasekaran et. al.. Adaptive Query Processing: Technology in Evolution. IEEE Data Engineering Bulletin, June 2000.

[10] P.J. Haas, J.M. Hellerstein. Ripple Joins for Online Aggregation. SIGMOD Conf. 1999: 287-298.

[11] J.M. Hellerstein, P.J. Haas, and H. Wang. Online Aggregation. SIGMOD Conf. 1997: 171-182.

[12] G.R. Hjaltason, H. Samet. Incremental Distance Join Algorithms for Spatial Databases. SIGMOD Conf. 1998: 237-248.

[13] Z.G. Ives, D. Florescu, M. Friedman, A.Y. Levy, and D.S. Weld. An Adaptive Query Execution System for Data Integration. SIGMOD Conf. 1999: 299-310.

[14] J.M. Patel, D.J. DeWitt. Partition Based Spatial-Merge Join. SIGMOD Conf. 1996: 259-270.

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

[16] J.M. Patel, J. Yu, and N. Kabra et. al.. Building a Scaleable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. SIGMOD Conf. 1997: 336-347.

[17] D. Rotem. Spatial Join Indices. ICDE 1991: 500-509.

[18] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith: The Sequoia 2000 Benchmark. SIGMOD Conf. 1993: 2-11.

[19] A.S. Szalay, P.Z. Kunszt, and A. Thakar et. al.. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. SIGMOD Conf. 2000: 451-462.

[20] H. Shin, B. Moon, and S. Lee. Adaptive Multi-Stage Distance Join Processing. SIGMOD Conf. 2000: 343-354.

[21] T. Urhan, M. Franklin. XJoin: Getting Fast Answers from Slow and Bursty Networks. Technical Report. CS-TR-3994, UMIACS-TR-99-13. February, 1999.