2014 IEEE 28th International Conference on Advanced Information Networking and Applications

# Efficient Distributed Spatial Semijoins and their Application in Multiple-Site Queries

Nawshad Farruque
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta
T1K 3M4 Canada
Email: nawshad.farruque@uleth.ca

Wendy Osborn
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta
T1K 3M4 Canada
Email: wendy.osborn@uleth.ca

***Abstract*—Applications exist today that require the management of distributed spatial data. Since spatial data is more complex than non-spatial data, performing distributed queries on it requires the consideration of both local processing (i.e. CPU and I/O) time and data transmission cost. To reduce these costs, one can use a distributed spatial semijoin as it eliminates unnecessary objects before their transmission to other sites and the query site. In this paper, we propose both new approaches for representing the spatial semijoin in a distributed setting, and their use for processing distributed queries consisting of any number of sites. We have tested our algorithms for four sites, which are a part of an actual working distributed system. We compare our algorithms with respect to data transmission cost, CPU time, I/O time and false positive results. We show that our algorithms are superior in many cases at optimizing the above criteria.**

## I. Introduction

Professionals from different application domains around the world must deal with the management and analysis of spatial data that is geographically distributed. One such application domain is in emergency and disaster management. One example is a distributed disaster management application for earthquake detection across many Canadian cities at risk [1]. Another example is a distributed emergency management system for Australia [5]. In both cases, distributing and storing the associated spatial data locally (as opposed to managing it centrally) will contribute greatly to real-time response in emergency situations. Therefore, spatial data has become an integral part of the world, and storing and querying it has become an important research subject.

Research in distributed spatial query optimization has focused on different distributed spatial data operations and distributed query optimization techniques to reduce the data transmission cost, CPU time and/or I/O time [11]. Approaches for processing distributed relational (e.g.alphanumeric) queries mostly focused on reducing the cost of data transmission, while considering the CPU and I/O time to be negligible [9]. However, due to the complex nature of spatial data, CPU and I/O costs must also be taken into consideration [11]. Most existing research explores the optimization of spatial operators, such as the spatial join or semijoin, in a distributed environment. Existing approaches can be grouped into distributed spatial join based approaches [6], distributed spatial semijoin-based approaches on a two-site or simulated multi-site system [7], [12], [8], and distributed Bloom filter approaches [13]. However, very few explore the use of these operators for processing a distributed spatial query that involves more than two sites. Exceptions to this [13], [8] use a simulated distributed environment or a parallel environment for evaluation.

Therefore, we explore new optimizations of the spatial semijoin in a distributed environment, and their use in a multi-site query processing strategy. We propose two strategies for compactly representing the spatial semijoin that reduce both the data transmission and local processing (CPU+I/O) costs when applied in a distributed spatial query. We utilize a Global Encompassing Minimum Bounding Rectangle (GEMBR), which is partitioned, mapped and applied in two different ways to approximate the objects in a spatial joining attribute. The first is partition indices, while the second is a Bloom filter [2], [3] representation. We apply each spatial semijoin in a multi-site distributed spatial query processing strategy. In addition, we also extend the two-site spatial semijoin proposed in [12] for multiple sites so that we have a benchmark strategy for comparison purposes.

We evaluate our query processing strategies in an actual (i.e. not simulated on one machine) distributed system, and show how our approaches outperform the extended spatial semijoin based strategy with respect to processing time and data transmission cost. We also compared our optimized approaches with respect to false positive rates.

## II. Background

In this section, we present background that is utilized in our strategies - specifically, on the R-tree, spatial semijoin, and Bloom filter. Our algorithm utilizes the R-tree [4]. The leaf nodes of an R-tree contain all the minimum bounding rectangles (MBRs) of the actual spatial objects, which are actually the tuples of a spatial relation. The non-leaf nodes contain the MBRs that encompass their child node MBRs.

A spatial join [11] combines tuples from two or more spatial data sets with respect to a spatial predicate. This predicate can be either a directional spatial relation and topological spatial relation. A spatial semijoin [12] is a modified spatial join operation which focuses on reducing data transmission costs. The semijoin achieves this by: 1) transferring only the spatial joining attribute and primary key from site 1 to site 2, 2) performing the spatial join of the spatial joining attribute from site 1 with the relation on site 2, and, 3) transferring only the relevant tuples from site 2 back to site 1, which are joined with the relation at site 1.

Many spatial operators can be optimized by applying the following two steps [11]: Filter and Refinement. In filter step the comparison between two spatial objects is evaluated using the approximations for each object, which are usually in the form of a minimum bounding rectangle (MBR). In the refinement step, the comparison between two spatial objects that pass the filter step is repeated with the actual spatial objects themselves.

The Bloom filter [2] is an array of $m$ bits which can be used to compactly represent a set of $n$ items and used for membership queries. In the context of query processing, a Bloom filter of $m$ bits can be used to represent a set of $n$ distinct attribute values. Given all $m$ bits initially set to 0, for each attribute value in $S$, the Bloom filter uses $k$ independent hash functions, each with the range $\{1, 2, ...., m\}$ to produce addresses for $k$ Bloom filter locations and to set the bit at each address to 1. To check if an attribute value $x$ is member of set $S$, $x$ is sent to the same hash functions to re-produce the addresses. If the bits at the re-produced addresses are set to 1, the $x$ is a a potential member of $S$.

However, because hash functions produce collisions, there is a certain probability of *false positives* occurring, which means that an attribute value can pass the Bloom filter test and not actually exist in $S$. The *false positive rate* is quantified in [3] as the following. Given that hash functions are random and all attribute values in $S$ have been processed, the probability of a bit being still 0 is:

$$p_{zero} = \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}} \tag{1}$$

Hence, the probability of *false positives* occurring is [3]:

$$p_{error} = (1 - p_{zero}) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k} \tag{2}$$

It is found that $p_{error}$ is minimum [3] when

$$k = \frac{m}{n} \ln 2 \tag{3}$$

So in our Bloom Filter Based Spatial Semijoin algorithm (BFSJ, see Section V below) to get the least possible false positives we have assumed the bloom bit factor ($\frac{m}{n}$) is 1.5, and the number of hash functions required is approximately 1.

## III. Related Work

In this section, we present a summary of existing strategies that process distributed spatial queries using spatial semijoins [12], [7], [8], Bloom filters [13], or space partitioning methods [10], [14]. Existing approaches in distributed spatial semijoins can be classified as: 1) modification to the operator itself [12], [7], and 2) general distributed query processing strategies [8].

Tan *et al.* [12] proposed two variants of the distributed spatial semijoin in order to speed up its processing. The authors represented the semijoin spatial attribute in two ways: MBRs that are obtained from an R-tree, and one dimensional ordered quad-tree-based locational keys. A performance evaluation of both semijoin operators found the following: 1) The R-tree and locational keys performed well when the data set is large, but locational keys performed better when the data set is small, 2) The R-tree performed much better than locational keys when CPU speed was high, and 3) Building an R-tree for this purpose incurred significant CPU cost. The main limitation of this work is that the evaluation took place on one machine only that simulated a two-site distributed database.

Karam and Petry [7] propose a distributed spatial semijoin operator that takes MBRs from different levels of the R-tree, instead of from the same level of the tree. A performance comparison versus the traditional distributed spatial join shows that their semijoin is superior when applied to real world data. However, CPU time is not considered, no comparison versus the distributed semijoin proposed in [12] is found, and no strategies that handle more than two sites were found.

Osborn and Zaamout [8] proposed a general distributed query processing strategy that utilizes MBR-based distributed spatial semijoins, and worked for queries that involved more than two sites. The strategy transmits the smaller spatial attributes to the sites that contain larger relations. After the semijoin is performed on those sites, the identifiers are then transmitted back to the originating (smaller) sites, and all qualifying tuples are sent to the query site for the final spatial join. An evaluation of their strategy with two-, four-, and six-site queries, found that the strategy has a lower data transmission cost - significantly lower in some cases - over the naïve spatial join approach. Limitations of this work include no consideration of CPU costs, the implementation and eavaluation on one machine only that simulated multiple sites.

Hua *et al.* [13] proposed a new spatial index structure, the BR-tree, which is an R-tree augmented with Bloom filters on every node that handle exact-match object queries. The leaf-node Bloom filters are created from the leaf node objects, while the non-leaf-level filters are created from its MBRs. Given a BR-tree at every site in the distributed spatial database, the central idea of the distributed query processing strategy is to distribute replicas of all BR-tree root nodes at all sites. Any object which is qualified by a root node is transmitted to the BR-tree that contains the original root node for further processing. One main limitation of this work are lack of support for spatial joins or spatial semijoin strategies. Another limitation is that it is unclear whether any significant geographical distribution existed between sites.

Patel and Dewitt [10] propose the Partition Based Spatial-Merge (PBSM) join, which handles the join of two spatial relations that do not have pre-existing spatial indices. Their strategy first partitions the **universe** that contains objects into multiple partitions, before forming buckets of spatial approximations from each relation that correspond to each partition. Then each pair of matching buckets (one per relation) are joined in memory using a plane-sweeping algorithm. Although our strategies are based on a partition of space, there are several differences between them and the PBSM strategy. First, the PBSM strategy is proposed for spatial joins without considering distribution across sites, while our strategies are semijoin based and designed for multi-site distributed queries. Second, it is unclear whether the same universe is assumed by

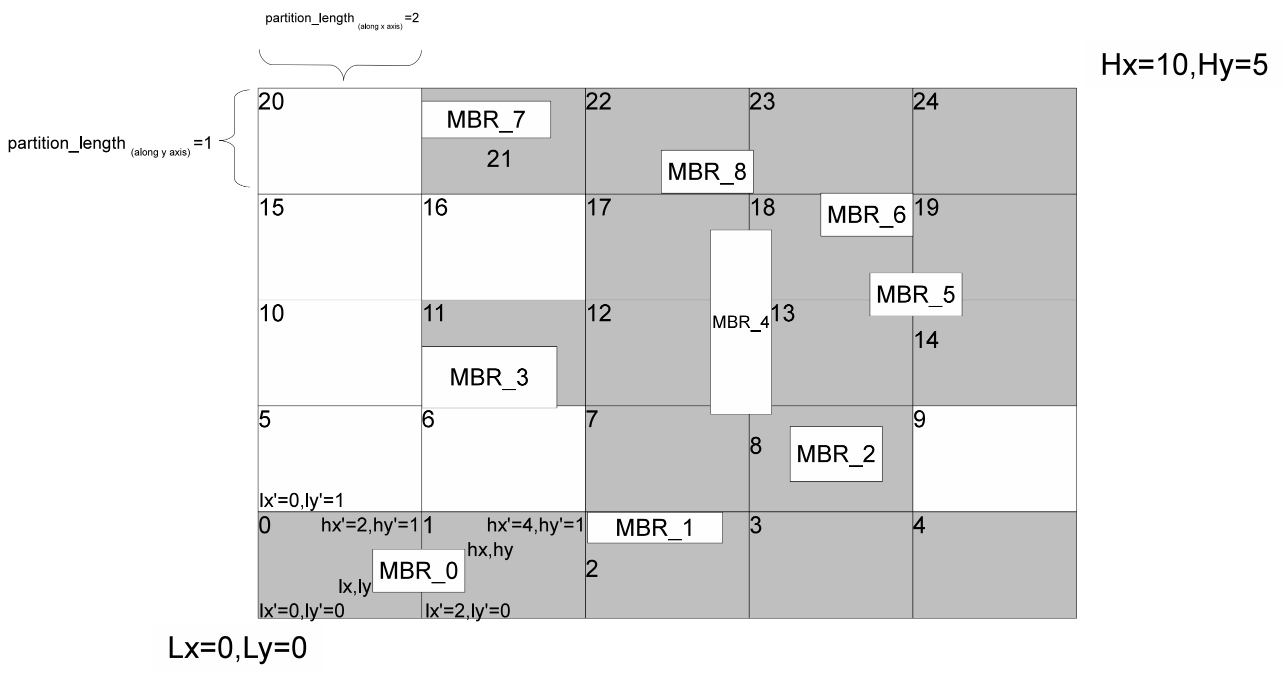

Fig. 1. GEMBR Partitioning and Mapping

both relations, or if the universe of one relation encompasses the other. Our strategy does not assume this. We determine the global space encompassing all objects as part of our partitioning (and our Bloom filter based) strategy. Finally, we transmit representations of the partitioned space to/from the query site, and only transmit to the query site tuples whose approximations qualify for the query. Our strategy does not form buckets from all approximations in order to perform a distributed spatial semijoin. We use the partition to filter approximations that do not participate in the final query.

Zhou *et al.* [14] extend the approach from [10] to work across parallel machines. Although CPU and communication costs are considered, as well as multiple sites, it is unclear how geographically distributed the sites involved are for the parallel strategy. Also, their strategy works with buckets formed from across all approximations, while again, our strategies only use the partition of space for compact representation purposes in a distributed spatial semijoin.

## IV. GEMBR Partitioning and Mapping

This section presents our core algorithms which are used by the compact representations of the distributed semijoins. First, the Global Encompassing Minimum Bounded Rectangle (GEMBR) is calculated. Then, the GEMBR is partitioned and all object MBRs are mapped onto it. Inside the GEMBR, all objects MBRs from all relations of all sites will reside.

### A. GEMBR Calculation

The GEMBR is the spatial extent of all objects that exist across all sites in the distributed spatial database. The steps for GEMBR calculation are as follows. First, the lower left and upper right coordinates (i.e. $(lx, ly)$ and $(hx, hy)$ respectively) of the Local Encompassing MBRs (LEMBR) are obtained from each of the sites. A Local Encompassing MBR (LEMBR) is the spatial extent of all spatial objects (or their MBRs) that exist at one site. If an R-tree exists at a site, this LEMBR can be extracted from the root node of the tree.

Then, the LEMBRs from each of the sites are sent to a query site. All the $(lx, ly, hx, hy)$ values of all the LEMBRs are divided into four sets – $lx$, $ly$, $hx$, and $hy$ – and each set is sorted. From these ordered sets, the lowest $(lx, ly)$ and the highest $(hx, hy)$ coordinates are identified. These are the resulting GEMBR coordinates.

For example, suppose we have four LEMBRs from four sites having the following $(lx, ly, hx, hy)$ coordinates: $(0, 0, 4, 2)$, $(1, 1, 6, 3)$, $(2, 2, 8, 4)$ and $(3, 3, 10, 5)$. After sorting the values in the $lx$ set in ascending order we have the following: 0, 1, 2 and 3. We take the first value, 0, which is the minimum $lx$ value among all values in the $lx$ set. Likewise, we also find $ly$, $hx$ and $hy$ values as 0, 10 and 5 respectively. So, the final GEMBR coordinate $(Lx, Ly, Hx, Hy)$ is $(0, 0, 10, 5)$. This example is shown in Figure 1.

After determining the GEMBR co-ordinates, it is sent to each of the sites in parallel. Then, using the partition information $n$ sent by the query site, the copy of the GEMBR at each client site is partitioned into $n \times n$ partitions, and indexed from lower left corner to upper right corner using positive integer, $i$, where $0 \leq i \leq n \times n$, which we call partition indices. Currently, the partition information $n$ is a constant value, which is stored at the query site, or can be provided by the user when specifying a query.

Then, the object MBRs at each client site are then mapped onto the partitioned GEMBR. If an R-tree is used, the object MBRs can be obtained from its leaf nodes, which results in lower I/O costs.

## B. GEMBR Partitioning and Mapping

After the GEMBR is calculated and transmitted to each site, GEMBR partitioning and mapping takes place on each site in parallel to partition the space and map the local object MBRs to the local GEMBR copy.

First, the GEMBR space is partitioned into $n$ partitions along the $x$ axis and $n$ partitions along the $y$ axis. The total number of GEMBR partitions is $n \times n$. The length of each partition along the $x$ axis and $y$ axis is calculated. Referring to the GEMBR example shown in Figure 1, where the GEMBR has the lower coordinate $(Lx = 0, Ly = 0)$ and upper coordinate $(Hx = 10, Hy = 5)$. Also, assume that the number of specified partitions $n$ along an axis is 5. We need to create $5 \times 5 = 25$ partitions. The length of each partition along $x$ axis is 2, and along the $y$ axis is 1.

Next, the coordinates $(lx\prime, ly\prime, hx\prime, hy\prime)$ for each partition is calculated. The process proceeds through the partitions in row-major order, starting from the lower left-hand corner $(Lx, Ly)$ to the top right-hand corner $(Hx, Hy)$ of the GEMBR. After each partition is calculated, it is stored in an array, which is called *partition coordinates holder array* (or *holder array* for short). The index values for the *holder array* will serve as the partition identifiers later on. Referring back to Figure 1, the lower left partition is calculated first (i.e $lx\prime = 0$, $ly\prime = 0$, $hx\prime = 1$, $hy\prime = 2$), followed by the next partition (i.e. $lx\prime = 2$,$ly\prime = 0$,$hx\prime = 4$,$hy\prime = 1$), and proceeding towards the upper right-hand partition, in row-major order.

Next, for each object MBR, the GEMBR partition (or subset of partitions, which we will call the GEMBR subregion) that encompasses the object is calculated. Because an MBR can overlap more than one partition, this step determines the region covered by the subset of partitions that contain an MBR.

For each object MBR, its lower left coordinate $(lx, ly)$ and upper right coordinate $(hx, hy)$, is tested against the lower left-hand and upper right-hand coordinates of each of the partitions, starting from the lower left-hand partition 0 and proceeding in row-major order to up the upper right-hand parition. If the lower coordinate $(lx, ly)$ is inside any partition, or on either of the partition coordinates ($lx\prime$,$ly\prime$) or ($hx\prime$,$hy\prime$), the lower left coordinates of a partition $(lx\prime, ly\prime)$ is recorded as the lower left-hand coordinate of the GEMBR subregion that encloses the object MBR. Similarly, the upper right coordinates ($hx\prime$, $hy\prime$) of a partition are recorded as the upper right-hand coordinate of the GEMBR subregion if it contains the upper right $(hx, hy)$ coordinate of an object MBR.

Referring back to Figure 1, suppose we test the lower left coordinates $(lx, ly)$ of $MBR\_0$ and find they are inside the lower left $(lx\prime = 0, ly\prime = 0)$ and upper right $(hx\prime = 1, hy\prime = 2)$ coordinates of partition 0 (i.e. its $(lx\prime, ly\prime) \leq (lx, ly) \leq (hx\prime, hy\prime)$). We record ($lx\prime = 0$,$ly\prime = 0$) from partition 0 as the lower left-hand co-ordinate of the GEMBR region containing $MBR\_0$. Then we check the upper right coordinate $(hx, hy)$ of $MBR\_0$ and find that it falls inside partition 1. We record as the upper right-hand coordinate of the GEMBR subregion the ($hx\prime = 4$,$hy\prime = 1$) coordinate from partition 1. Therefore, these lower left-hand and upper right-hand coordinates define the subregion of partitions that encompass $MBR\_0$.

Finally, for each object MBR and its corresponding subregion found in step 3, all the partition indices of the partition (or partitions that fall within the GEMBR subregion) are identified by comparing each subregion with the partitions in the holder array. The set of unique partition indices from all object MBRs are returned for the final mapping result.

Referring back to Figure 1, we identify the partitions that contain $MBR\_0$, using the region defined by ($lx\prime = 0$, $ly\prime = 0$) and ($hx\prime = 4$, $hy\prime = 1$) and the *holder array* of partitions. This results in partitions 0 and 1 for $MBR\_0$. Therefore, partition indices 0 and 1 are returned for $MBR\_0$. Similarly, $MBR\_1$ is mapped to partition indices 2 and 7, $MBR\_2$ to partition index 8, $MBR\_3$ to partition index 11, $MBR\_4$ to partition indices 7,8,12,13,17 and 18, $MBR\_5$ to partition indices 13,14,18 and 19 and $MBR\_6$ to partition indices 18,19,23 and 24, $MBR\_7$ to partition index 21 and $MBR\_8$ to partition indices 22 and 23.

## V. Query Processing Strategies

In the following section we propose three distributed spatial query processing algorithms, two of which utilize our compact representations of the distributed spatial semijoin: Geometric Space Partition and Mapping Based Spatial Semijoin (PMSJ), Bloom Filter Based Spatial Semijoin (BFSJ), and Distributed Naïve Spatial Semijoin (NSPJ).

Our algorithms are designed to work for any number of sites. Among the sites, one is designated as the query site where the user issues a query. All other sites are client sites which process a portion of the user query. All the processes initiate at each site at the same time when the user issues query from query site. In the user query the user states the number of partitions *n* along any one axis of GEMBR for both algorithms. For the BFSJ algorithm *bloom filter factor* and *number of hash functions* are also stated.

### A. Geometric Space Partition and Mapping Based Spatial Semijoin (PMSJ)

Our PMSJ algorithm performs optimization of distributed spatial queries by utilizing the partition indices representation of the GEMBR from all participating sites for distributed semijoin processing. First, on each client site, the partition indices are obtained from the GEMBR Calculation, Partition and Mapping algorithms, and duplicates are removed before transmission to the query site. This is shown in Figure 2.

Then, at the query site, the set of common partition indices is calculated. A partition index is added to this set only if it was sent from every client site (i.e. at every client site, the partition contained one or more objects). Then, the final set of common indices are sent back to each client site. This is shown in Figure 3.

On each client site, all the tuple ids of the corresponding object MBRs that reside in the partitions contained in the set of common partition indices are retrieved. Finally, on each client site, for each qualifying tuple id, the corresponding exact spatial object are retrieved and sent to the query site for the refinement step. This is depicted in Figure 4. Note that in Figure 4 each spatial object is represented with its tuple id due to limited space in the diagram - however, it is the objects that are being transmitted to the query site.

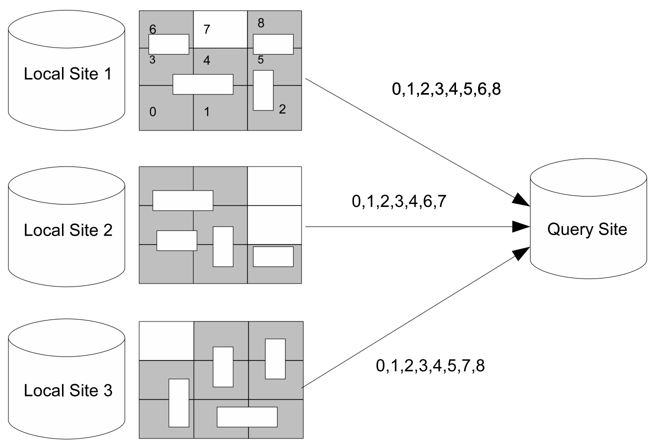

Fig. 2. Sending Partition Indices to Query Site

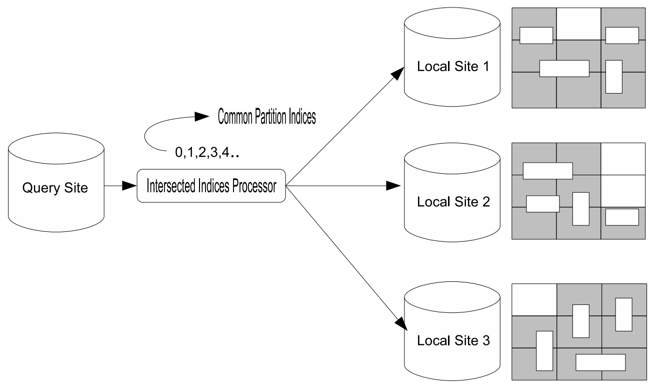

Fig. 3. Calculation and Transmission of Common Partition Indices

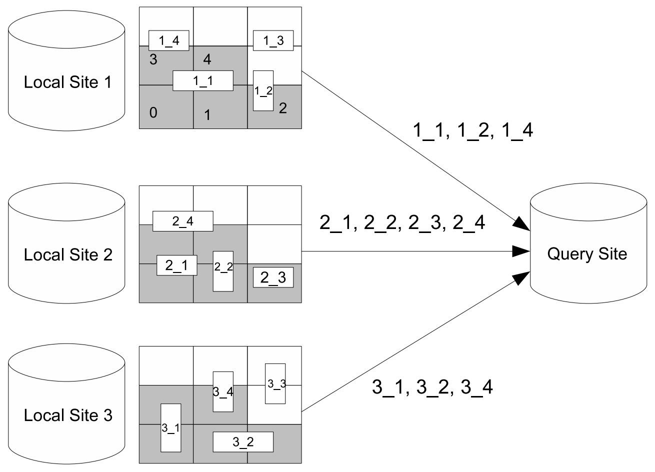

Fig. 4. Object Transmission to Query Site

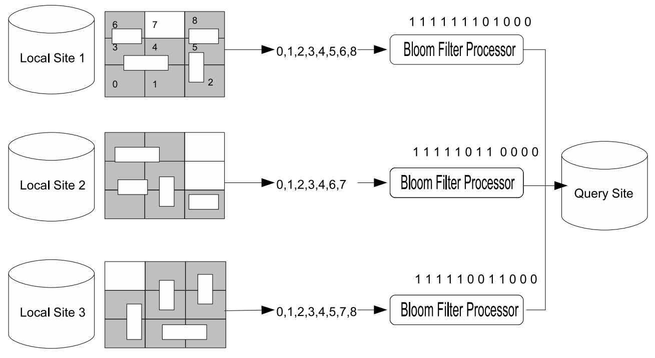

Fig. 5. Mapping Partition Indices to Bloom Filter

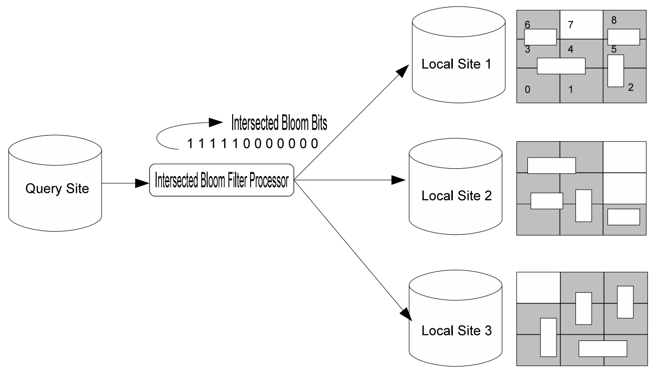

Fig. 6. Calculation and Transmission of Common Bloom filter

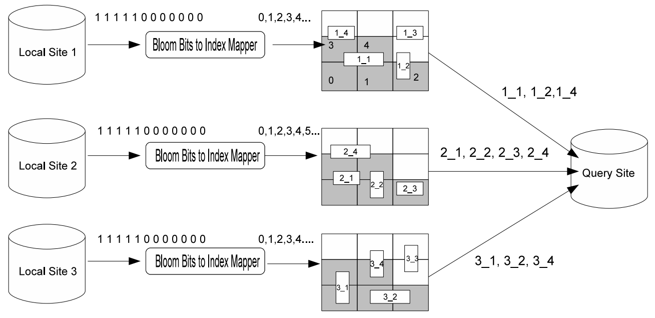

Fig. 7. Bloom Bit to Partition Index Mapping and Object Transmission to Query Site

### B. *Bloom Filter Based Spatial Semijoin (BFSJ)*

Our BFSJ algorithm creates and uses Bloom filter-based representations of the GEMBRs from all participating sites for semijoin processing. Given the query information, which consist of the number of the partitions along each axis $n$, the number of hash functions $k$ and the Bloom filter factor $B_f$, the size of the Bloom filter is calculated based on following formula: $B_s = B_f \times (n \times n)$ at each of the client sites. Then all the Bloom filters are initialized to all 0s.

Then, the Bloom filter representations of the partition indices from each site are constructed and sent to the query site. Bloom filter creation is done by a module called the *Bloom Filter Processor*. For this, the *Bloom Filter Processor* takes the set of partition indices that are returned from the GEMBR Calculation, Partitioning and Mapping functions, and sends it to the hash functions. The hash functions calculate and return the corresponding Bloom filter indices, which are then set to 1. This is shown in Figure 5.

At the query site, the *Intersected Bloom Filter Processor* module finds the common Bloom filter by performing a bit-wise intersection of all Bloom filters in order to find out the common bloom bits from all the bloom bits representations and send the common or intersected bloom bits to each of the client sites. This is depicted in Figure 6.

At each client site, the common Bloom filter bits are sent to the *Bloom Bits to Index Mapper* module, which maps the bits to the corresponding partition indices. This is done by taking each partition index and if it is found to be 1 in the Bloom filter (after being hashed by all the hash functions), it is kept. These partition indices are eventually the common partition or intersected partition indices.This is shown in Figure 7.

At each client site, all the tuple ids of the corresponding object MBRs that reside in the partitions contained in the set of common partition indices are retrieved. Finally, on each client site, for each qualifying tuple id, the corresponding exact spatial object are retrieved and sent to the query site for the refinement step. This is also depicted in Figure 7. Note that in Figure 7 each spatial object is represented with its tuple id due to limited space in the diagram - however, it is the objects that are being transmitted to the query site.

### C. *Distributed Naïve Spatial Semijoin (NSPJ)*

The Distributed Naïve Spatial Semijoin(NSPJ) algorithm is an extension of [12] for more than two sites. We use this as a benchmark strategy for comparison purposes.

All the object MBRs at each client site are sent to the query site. If the spatial relation at a client site is indexed by an R-tree, then the object MBRs can be obtained by scanning the leaf nodes of the R-tree. At the query site, all object MBRs from all sites are checked for overlap. All qualifying object MBRs are sent back to their respective client sites and their corresponding tuple ids are extracted. Then, all the qualifying exact spatial objects for those tuple ids are sent to the query site for the refinement step.

## VI. Performance Evaluation

In this section we present our performance evaluation of the BFSJ, PMSJ and NSPJ strategies. Our first goal is to compare the BFSJ and PMSJ strategies against the NSPJ strategy with respect to processing and data transmission costs. Our second goal is to compare the BFSJ and PMSJ strategies with respect to false positives.

We have implemented our algorithms in a four node peer-to-peer distributed system[1]. The nodes are situated in four geographically scattered locations. Our query site is located at the University of British Columbia (node name: Orcinus) and the client sites are located at the University of Victoria (node name: Hermes/Nestor), Simon Fraser University (node name: Bugaboo ) and the University of Alberta (node name: Jasper)[2]. We have also used a parallel distributed shell (PDSH) utility[3] for co-ordinating the overall spatial semijoin process in parallel at each client site, as soon as the user issues the query from the query site. We have implemented all our algorithms in C++ and organize the total process to be done in the distributed environment with the help of Bash shell scripting.

For our evaluation, we used randomly-generated synthetic data. We chose to use synthetic data so that results and trends could be determined for specified numbers of tuples and a random data distribution. Each client site contains five spatial relations that contain 2000, 4000, 6000, 8000 and 10,000 tuples respectively. Every spatial relation contains one spatial attribute, which contains a 10-unit by 10-unit square. For each relation, a space size of $\sqrt{\#tuples * 10}$ units is used to contain all randomly generated objects. Each relation is indexed on its spatial attribute with an R-tree[4].

We compare the BFSJ, PMSJ and NSPJ algorithms with respect to the processing (CPU+I/O) time and data transmission cost. The CPU+IO time was measured in seconds, while the data transmission cost was measured in the number of kilobytes that were transmitted over the network. We also compare the BFSJ and PMSJ algorithm based on the percentage of false positives. As the NSPJ uses the object MBRs directly for overlap checking, it does not produce any false positive result in the filter stage. Therefore, we have only calculated the percentage of false positives for BFSJ and PMSJ. The percentage of false positives is calculated based on the following formula:

$$f = \frac{\#Tuples_{BFSJorPMSJ} - \#Tuples_{NSPJ}}{\#Tuples_{BFSJorPMSJ}} \times 100 \quad (4)$$

### A. *Results*

We show comparison of processing time (PT items in each legend) and average transmission cost (TC items in each legend) for 30x30, 60x60 and 90x90 partitions in Figures 8, 9, and 10 respectively. We also show the false positive percentage comparison in Figures 11, 12 and 13 for the same. From the figures, we observe a common trend of increasing transmission cost and increasing processing time while decreasing false positive percentages with the increase in the number of partitions.

*1) Transmission Cost:* We calculate the average transmission cost in kilobytes (our "inefficiency index" measure for

[1]Westgrid, www.westgrid.ca

[2]https://www.westgrid.ca/support/systems

[3]PDSH utility obtained from code.google.com/p/pdsh

[4]R-tree code obtained from www.rtreeportal.org.

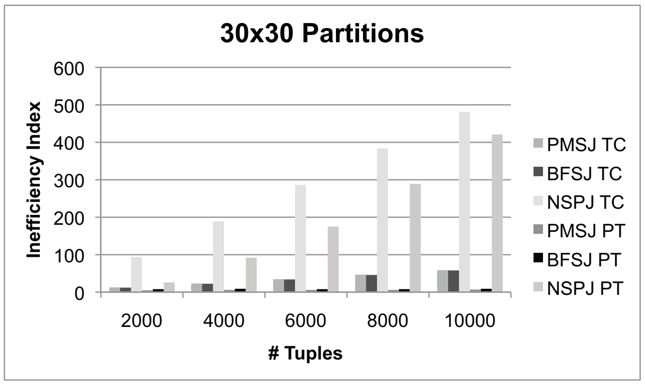

Fig. 8. Processing Time and Transmission Cost for 30 × 30 Partitions

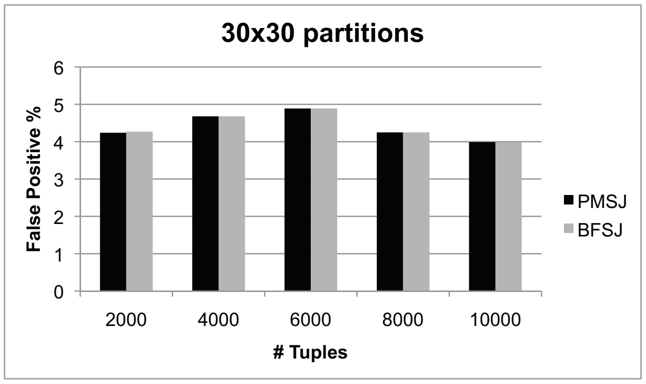

Fig. 11. False Positives Comparison for 30 × 30 Partitions

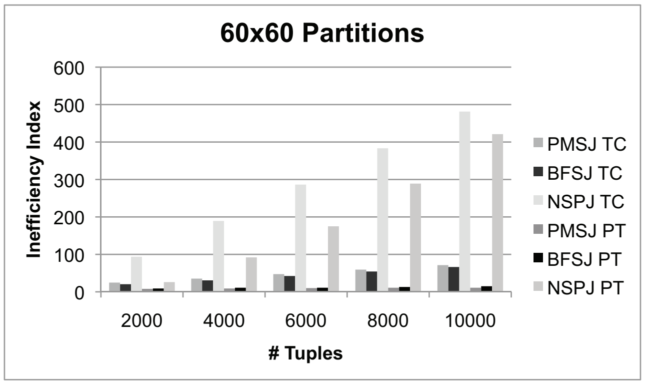

Fig. 9. Processing Time and Transmission Cost for 60 × 60 Partitions

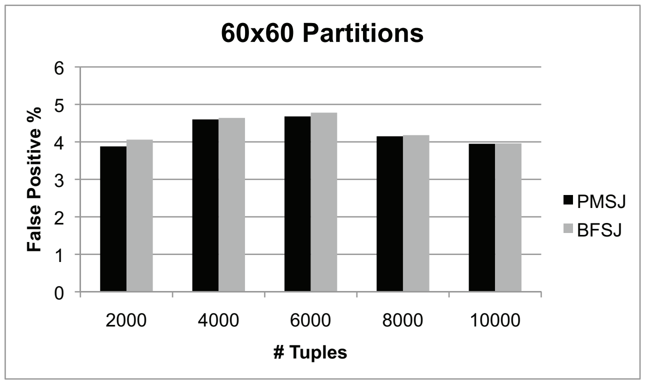

Fig. 12. False Positives Comparison for 60 × 60 Partitions

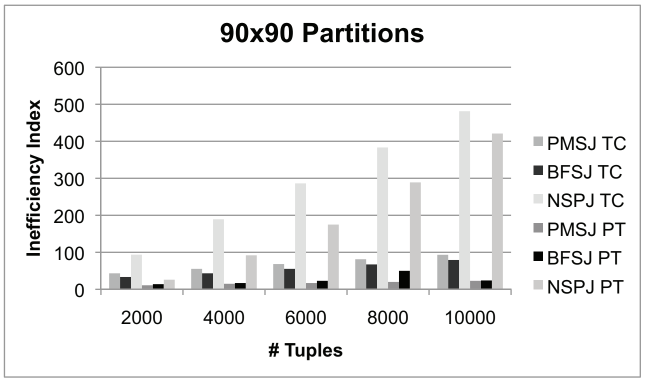

Fig. 10. Processing Time and Transmission Cost for 90 × 90 Partitions

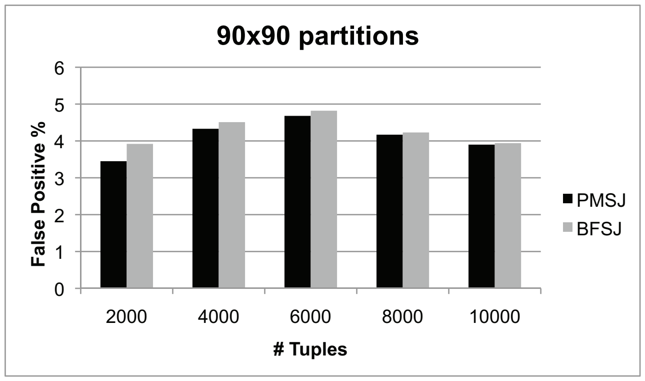

Fig. 13. False Positive Comparison for 90 × 90 Partitions

transmission cost, as displayed in the Figures) of data transmitted from client sites to query site and vice-versa in parallel. We plot this data against the number of tuples. By comparing the algorithms we find that the PMSJ and BFSJ algorithms outperform the NSPJ algorithm by a factor of approximately 6 on average with respect to transmission cost. For example, in Figure 10, we see a significant difference in transmission costs between both the BFSJ and PMSJ algorithms and the NSPJ algorithm. This cost for NSPJ is very high with respect to number of tuples. This observation is true as well for Figures 8 and 9. The reason is that we send more compact approximations of MBRs (i.e. Bloom filters and partition indices) to and from the query site in our algorithms, than in NSPJ where we send the actual MBR approximations of the spatial objects. We further reduce the transmission cost by sending unique partition indices which are covered by object MBRs on a client site to the query site for semijoin processing or to the *Bloom filter processor* for Bloom filter creation.

We also compare both the BFSJ and PMSJ algorithms and observe that these algorithms perform very closely, with BFSJ having a transmission time gain of approximately 1.12 over PMSJ. This is due to sending Bloom filters, which only contain boolean values and are more compact than the partition indices. This is seen in Figure 8 for $30 \times 30$ partitions where both the strategies almost tied and in Figures 9 and 10 respectively where we can see a little performance gain of BFSJ over PMSJ for $60 \times 60$ and $90 \times 90$ partitions. We see that overall transmission cost increases linearly with the increase of partitions (transmission cost increases roughly 1.9 times when partition number increases three times) and number of tuples(transmission cost increases 3.3 times when number of tuples increases five times) in BFSJ and PMSJ. This happens because more partitions results in more data transmission. This observation is evident in Figures 8, 9 and 10.

*2) Processing Time:* For processing time we calculate the amount of time the algorithms take in seconds (our "inefficiency index" measure for processing cost, as displayed in the Figures) which includes both CPU and I/O times. We plot this against the number of tuples in Figures 8, 9 and 10. We find both BFSJ and PMSJ are on average 18 times faster than NSPJ for $30 \times 30$, $60 \times 60$ and $90 \times 90$ partitions. For example Figure 8 shows that there is a significant difference between BFSJ, PMSJ and NSPJ. This is because in the BFSJ and PMSJ algorithms, we use more compact representations for semijoin processing, which are just integers or boolean values, but in NSPJ we are using the actual object MBRs for the same, which incurs extra CPU time.

We see very little performance gain of PMSJ over BFSJ, which is roughly on average 1.38. This is because there is extra processing for creating and setting bits in the Bloom filters, and remapping partition indices using the Bloom filters. This observation is consistent for $60 \times 60$ partitions shown in Figure 9 and $90 \times 90$ partitions shown in Figure 10. The increase of the processing time is linear with respect to the increase in number of partitions, as the semijoin operates on more data. For example, if we increase the number of partition three times, the increase of processing time is 2.9 times. This is also clearly observed in the Figures 8, 9 and 10.

*3) False Positive Comparison:* With respect to the false positive percentages, we find that the difference between BFSJ and PMSJ is more prominent with the increase in number of partitions. For example, there is a tie between PMSJ and BFSJ for $30 \times 30$ partitions, which is depicted in Figure 11. However, the superiority of PMSJ becomes significant when the number of partitions increases as depicted in Figure 12 and Figure 13. PMSJ is always ahead of BFSJ by a factor of 1.02 on average, which we observe from the little performance gain of PMSJ over BFSJ we see in Figures 11,12 and 13. With the increased number of partitions, the false positive percentage is linear with a gain of 1.02 (which means the false positive percentage decreases) on average for both of the PMSJ and BFSJ algorithms which is also observed in all the three Figures. With more partitions we obtain more accurate mapping of object MBRs in GEMBR and thus the false positive percentage decreases. PMSJ predominates in this case over BFSJ because, BFSJ takes the partition indices as its input so it should have at least the false positive percentage as the PMSJ with the false positive hits of its own.

## VII. Conclusions

In this paper, we propose two representations of the spatial semijoin and their use in multiple-site distributed spatial query processing strategies. To further reduce the data transmission and processing costs, we converted the geometric representation of object MBRs to simple integers or binary bits. We evaluated our algorithms in a real life peer-to-peer distributed system with synthetic data sets. We found that our optimized algorithms perform significantly better than the naïve strategy.

In the future, we are looking towards testing our system against a larger number of distributed nodes. We are also looking for improved strategies for partitioning the GEMBR space to gain improvements in processing speed. In addition, we will explore more compact representations of the object MBRs for efficient semijoin processing. Finally, we will test our BFSJ algorithm by using different types of hash functions and number of bloom filters and analyze the performance.

## VIII. Acknowledgments

We are grateful to WestGrid and Compute Canada-Calcul Canada (www.westgrid.ca) for the use of their resources.

## References

[1] R. Abdalla and V. Tao. Integrated distributed GIS approach for earthquake disaster modeling and visualization. In *Geo-Information for Disaster Management*, pages 1183–1192. 2005.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[3] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[4] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[5] T. Hunter. A distributed spatial data library for emergency management. In *Geo-Information for Disaster Management*, pages 733–750. 2005.

[6] M.-S. Kang, S.-K. Ko, K. Koh, and Y.-C. Choy. A parallel spatial join processing for distributed spatial databases. In *Proceedings of the 5th International Conference on Flexible Query Answering Systems*, pages 212–225, 2002.

[7] O. Karam and F. Petry. Optimizing distributed spatial joins using R-trees. In *Proceedings of the 43rd ACM Southeast Conference*, 2006.

[8] W. Osborn and S. Zaamout. Multiple-site distributed spatial query optimization using spatial semijoins. In *Proceedings of the 10th International Baltic Conference on Databases and Information Systems*, pages 11–19, 2012.

[9] M. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 2011.

[10] J. Patel and D. DeWitt. Partition based spatial-merge join. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 259–270, 1996.

[11] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice Hall, New Jersey, 2003.

[12] K.-L. Tan, B. Ooi, and D. Abel. Exploiting spatial indexes for semijoin-based join processing in distributed spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(6), 2000.

[13] H. Y, B. Xiao, and J. Wang. Br-tree: A scalable prototype for supporting multiple queries of multidimensional data. *IEEE Transactions on Computers*, 58(12):1585–1598, 2009.

[14] X. Zhou, D. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, 2:175–204, June 1998.